COMPUTER-AIDED DIE DESIGN FOR PLASTIC PRODUCTS

by

TRUONG TUAN GIAC

B. E. (Hons.)

This Thesis is submitted for the Degree of Doctor of Philosophy in Mechanical Engineering at the University of Canterbury, Christchurch, New Zealand.

University of Canterbury

February 1978.

CONTENTS

ACKNOWLEDGEMENTS

I am extremely thankful to Dr. K. Whybrew of the Mechanical Engineering Department, who initiated the project, for his constant guidance, encouragement and kindness. I am also deeply indebted to Professor H. McCallion for his criticisms and confidence. His patience in reading my successive drafts has enabled me to present my work in clear, understandable English while for his part, he was only reminded of the Victorian lady.

Among the engineers, I would like to thank particularly Mr. D. Harris of P.D.L. Industries Ltd. and Mr. P. Neill of P.D.L. Plastics Ltd. for their discussions and demonstrations on the practical aspect of die design.

I would also like to thank Mr. R. Harrington and Mr. A.D. Causer for their help in the field of computer systems and computer programming techniques.

Last but not least, I would like to express my thanks to all my fellow students, my dear friends, the staffs of the Mechanical Engineering Department, the Computer Centre and the P.D.L. Industry groups, who have been giving me continuous support throughout my work.

SYNOPSIS

In designing a die for moulding plastic products, a great proportion of the work is very tedious because it does not require any creative effort. The very first step in building up a general die assembly involves many creative ideas and skills but once this is done, the next step, the production of engineering drawings for individual die components, is a routine matter of data extraction because all components can be derived from the assembly. This latter step is repetitive and time consuming. These factors make it very prone to human errors. The purpose of this project has been to analyse the design process with a view to giving computer assistance wherever possible. Based on this analysis an interactive computer-based design process is proposed. The approach has been pragmatic and the process designed so that it can be followed by a designer skilled in the conventional design process. The programs, written in Fortran, were designed for use with the GT-44 graphics unit on-line with the 24K PDP-11 computer.

The major difficulty has been the three dimensional description of the die cavity and die components. Various description methods were studied and in general, the method chosen for a particular application depends very much on the geometry of the objects under consideration. They generally involve piece-wise representation of surfaces e.g. Coon's patches, Bézier and Ferguson patches. These were deemed to be unnecessarily general for this application and prohibitively extravagant in their use of computer time and storage. Consideration of typical dies for plastics showed that they were usually constructed from simple geometrical shapes with flat, cylindrical and conical surfaces. A method, called the block-building method, was developed. In this

method, a library of primitive objects having simple shapes was defined. Primitive objects consisted of: rectangular blocks, cylinders, truncated cones, truncated pyramids, wedges and fillets. These primitive objects can be used to build the required object in the same way as houses are built from simple, rectangular bricks. While I was developing the method independently at this university, I.C. Braid published his first paper, "The Synthesis of Solids Bounded by Many Faces", describing the same principle. His work has been referred to extensively in this project although it was limited to a smaller class of object shapes and differed in other details.

Objects were built by suitable additions and subtractions between primitive objects or other objects. To do this, there were two algorithms called merging and intersection. Merging merges two objects together provided that they have a pair of flat, coplanar faces. When two positive objects are merged the action is similar to cementing them at their coplanar faces. However, when a negative object which is completely inside a positive object is merged into it the result is to remove the part of material having the shape of the negative object from the positive object. Intersection finds the object resulting from the intersection of a negative object and a positive object. The two original objects do not necessarily have to have any pair of coplanar faces. The resulting object has all the material inside the positive object but outside the negative object. Intersection can also be used to find the union of two positive, intersecting objects. In this case, the two objects are negated first, then intersected and finally negated again.

To conclude the project, a very simple example for a compression die was given to demonstrate the feasibility of the computer-aided design process.

SUMMARY

This work was aimed at building up a computer system to assist a plastic die designer through his design process, a great percentage of which has been found to be of a routine nature. The computer graphics system consisted of a GT-44 graphics unit (supplied by Digital Equipment Corporation Ltd.) driven by a PDP-11 computer with a 24k-word memory. All programs were written in RT-11 Fortran language. The system was designed to be interactive, which means the user is stopped at different stages and asked for his decision on problems which only human judgement can solve. Only compression and injection moulding processes were considered.

Chapter 1, the introduction, states the problem. The process through which a conventional designer would go to design a compression or an injection die is postulated. It is then pointed out which stages could be automated i.e. computerized and which stages required human judgement. From this consideration, the question of how to describe an object in three dimensions emerged.

Chapter 2 presents various description methods. In general, each method was adopted to suit a certain class of object shapes and they all proved to be unnecessarily general. Examination of typical plastics dies showed that they were usually made of simple shapes bounded by flat, cylindrical and conical surfaces. From this consideration the block-building technique was developed. This method involved setting up a library of primitive objects having simple shapes: a box, a cylinder, a wedge, a fillet, a truncated cone and a truncated pyramid. A general object shape could then be built up through suitably adding and subtracting the primitive objects. Object addition and subtractions were described in two algorithms: merging and intersection. The coincidence of this method and the one employed by Braid [1] was emphasised. Braid's

work was limited in the shapes of the object (conical surfaces were not allowed) and differed in details.

Chapter 3 gives the description of six primitive objects. Primitive objects were the basic building units which when assembled in a certain way made up a class of general objects. Objects could be made positive or negative. A positive object is a solid object while a negative object is an empty object. An extension to conical surfaces was made here compared with Braid.

The next two chapters: Chapters 4 and 5, are devoted to merging and intersection respectively. These algorithms allowed primitive objects to be combined together to build up a new object as their names suggested. Two non-intersecting objects which have at least one pair of flat co-planar faces can be merged. When two positive objects which are outside each other are merged, they are effectively joined at their coplanar faces. When a negative object which lies completely inside a positive object is merged into it, the result is to remove the shape of the negative object from the positive object. The characteristics of merging are that no new edges are created and when an original face is deleted it is always replaced by a new face lying on the same plane. The basic intersection algorithm finds the object resulting from the intersection of a positive object and a negative object. The two objects do not necessarily have to have any coplanar faces. The resultant object has all the material inside the positive object but outside the negative object. However, intersection can also be used to find the union of two positive intersecting objects. To do this, the objects are negated first then intersected and finally negated again. The characteristics of intersection are that new edges can be created and an original face can be completely deleted without being replaced by any new faces.

Chapter 6 gives the description of the implementation of the system

with an emphasis on the severe restriction imposed by the small size of the core. The system had to be written in various separate programs, each of them could be run at any stage as the user wished. All programs used the overlay structure to increase the effective size of the core. The immediate data between programs was stored and saved on a disc file whose structure was also given.

Chapter 7 is devoted to a simple compression die design as an example to prove the feasibility of the system and in particular to illustrate the design process of Chapter 1.

The project was concluded by chapter 8 containing the conclusions and a description of future work.

CHAPTER 1

INTRODUCTION

1.1 Statement of the Problem

Although plastics products have been widely used, due to their many benefits, in both industrial and domestic fields, yet there has been very little progress made to approach a scientific design procedure for designing the moulds. In most cases, the designer has to recall his or others past experience as well as using his own judgement, or even guessing, in making a decision. It is well known by plastics mould designers that once the general arrangement of a die has been decided, a creative process requiring a great deal of skill, knowledge and ingenuity, the designer has to spend a long time producing detailed drawings of die parts, a routine process requiring patience, arithmetic, and attention. The second process which may take the designer weeks to complete is ideally suited to the type of work of a computer and, in particular, computer graphics system. Computer graphics not only produce drawings more economically, efficiently and accurately, but also offer a very good means of man-machine communication.

The objectives of this project are to employ computer graphics to:

1. Take over the entire work load from the designer in the second process of his design. An investigation into the works already done by specialists shows that computer graphics systems are capable of:
   (a) designing, storing, displaying all 3-dimensional objects
   (b) transforming an object under any required transformation
   (c) efficiently processing data associated with objects, thanks partly to their well-designed data structures.

These capabilities are sufficient to produce detailed engineering

drawings of die components once a general assembly of a die has been made and stored.

2. Help the designer to sketch and analyse all possible design alternatives of the general assembly of a die. This process when done manually is very time-consuming and tedious.

A typical design is a creative work which is usually a combination of the designer's practical experience and a theoretical analysis. Unfortunately, the latter is often too complex and hence a compromise must be made. Human judgement is therefore essential and is a field where a computer cannot replace a human being. The nature of this system is interactive, a process in which the designer has to make decisions at different steps about how and what to do next and the computer acts accordingly. To have a closer look at how this is done, let us look at the proposed design procedure outlined below.

1.2 The Proposed Design Procedure

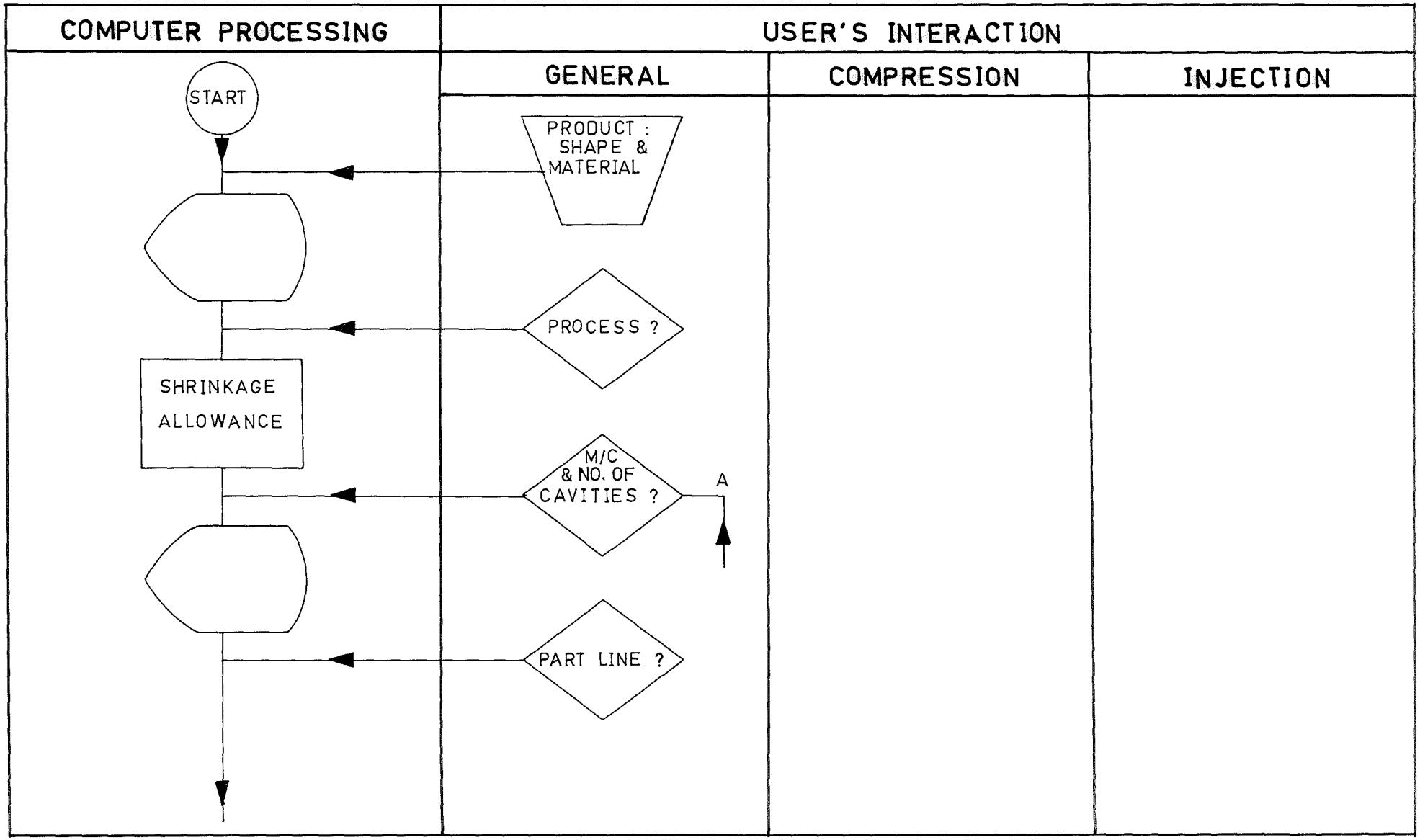
A proposed design procedure for a compression and an injection mould, with its possible computer interaction, is outlined in Fig. 1.1. The steps in this process are:

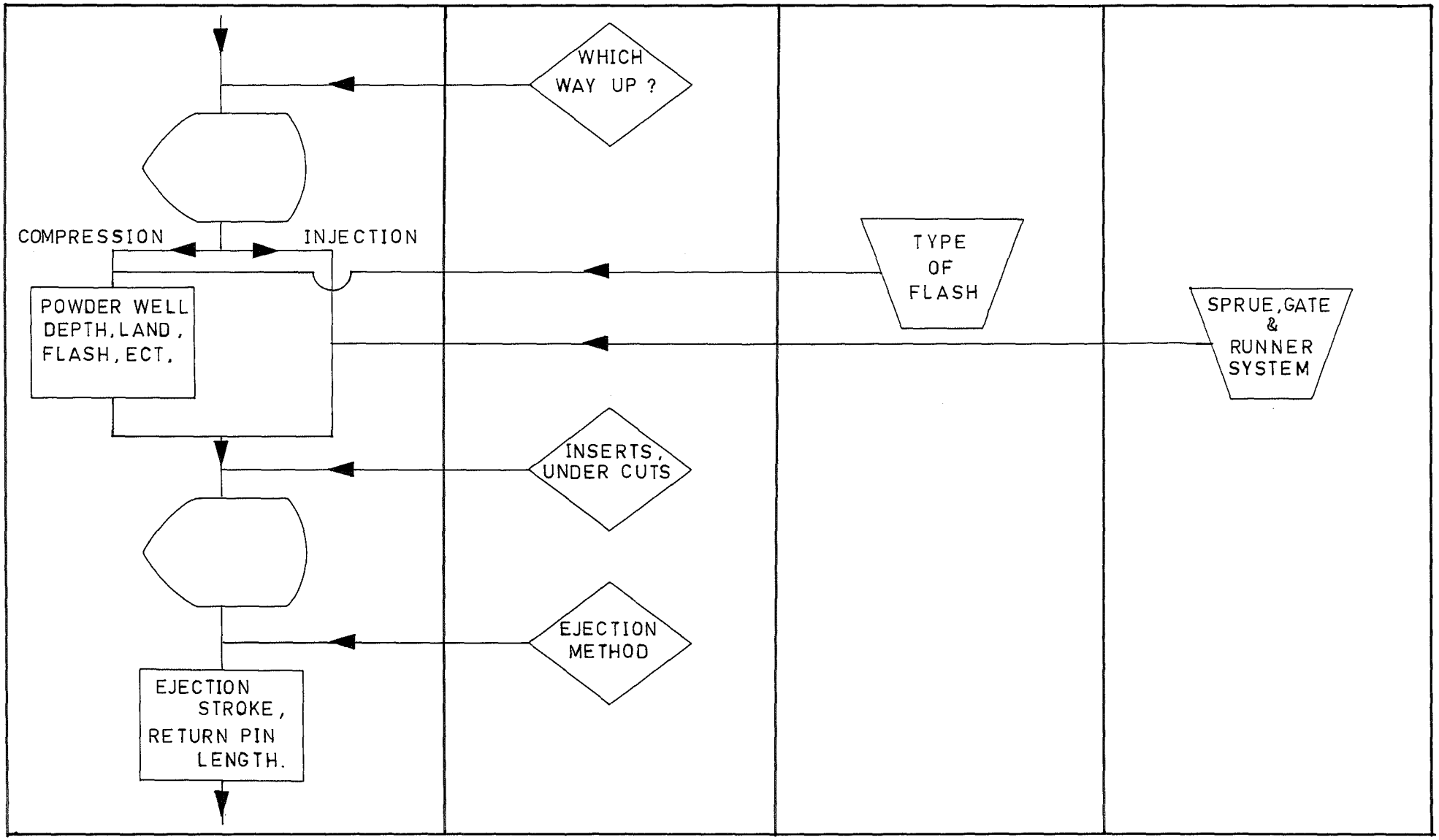
(1) Product Shape and its Material

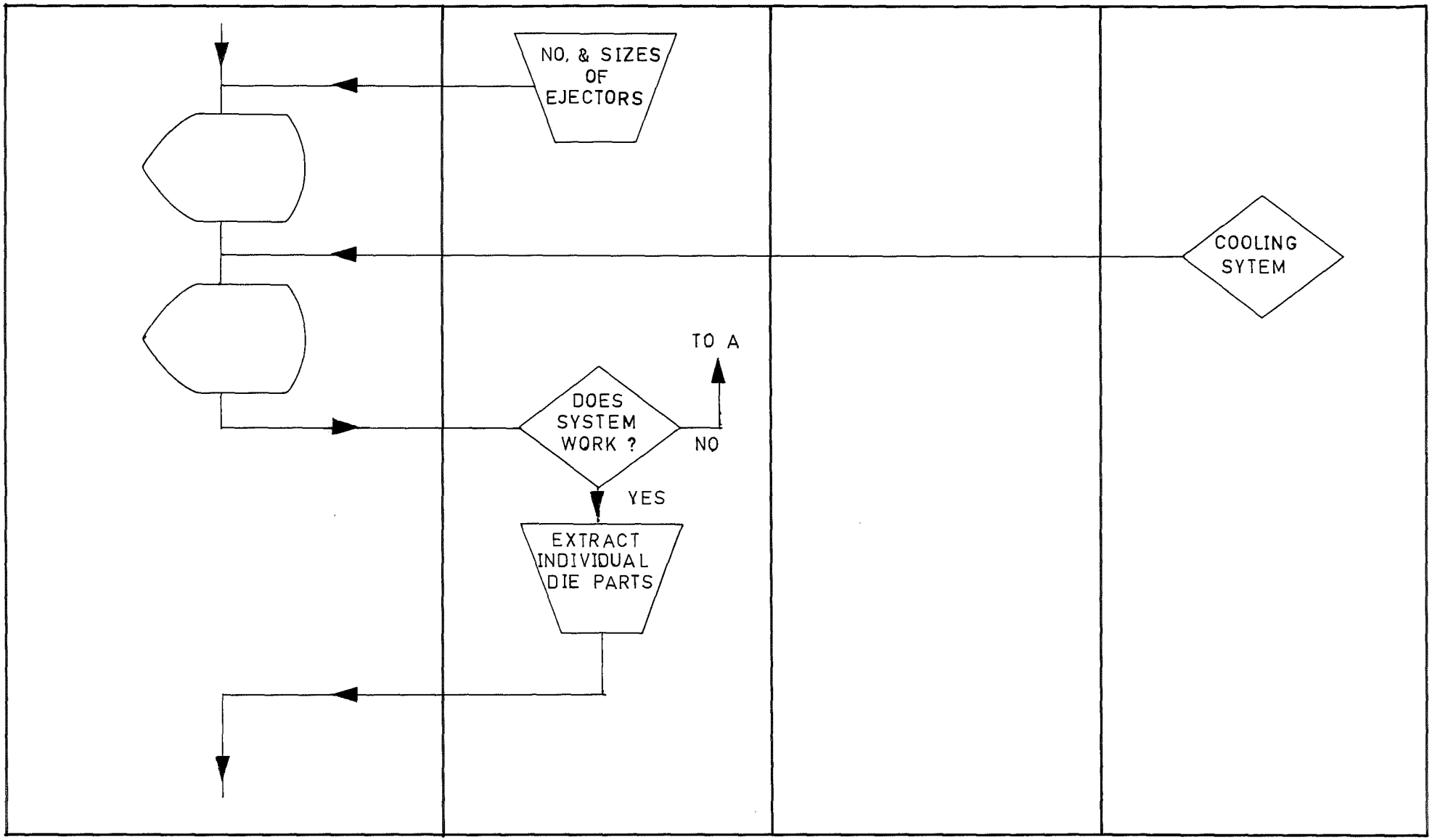
This is the first step in the design process. The die designer receives complete drawings of the product with specified material to be produced. He can feed this information into the computer in such a way that it can interpret the shape of the product. The problem of object description therefore emerges, which will be further discussed later. The shape of the product is now described and stored on a computer device, say, a disk.

(2) Which process is best to be used to produce the product

The first question that comes to the designer's mind is how the product is to be moulded. The decision is dependent on the shape


COMPUTER PROCESSING
USER'S INTERACTION
GENERAL
COMPRESSION
INJECTION
START
PRODUCT : SHAPE & MATERIAL
PROCESS ?
SHRINKAGE ALLOWANCE
M/C & NO. OF CAVITIES ?
A
PART LINE ?


WHICH WAY UP ?
COMPRESSION
INJECTION
TYPE OF FLASH
SPRUE, GATE & RUNNER SYSTEM
POWDER WELL DEPTH, LAND, FLASH, ECT.
INSERTS, UNDER CUTS
EJECTION METHOD
EJECTION STROKE, RETURN PIN LENGTH.


NO, & SIZES OF EJECTORS
COOLING SYTEM
TO A
DOES SYSTEM WORK ?
NO
YES
EXTRACT INDIVIDUAL DIE PARTS

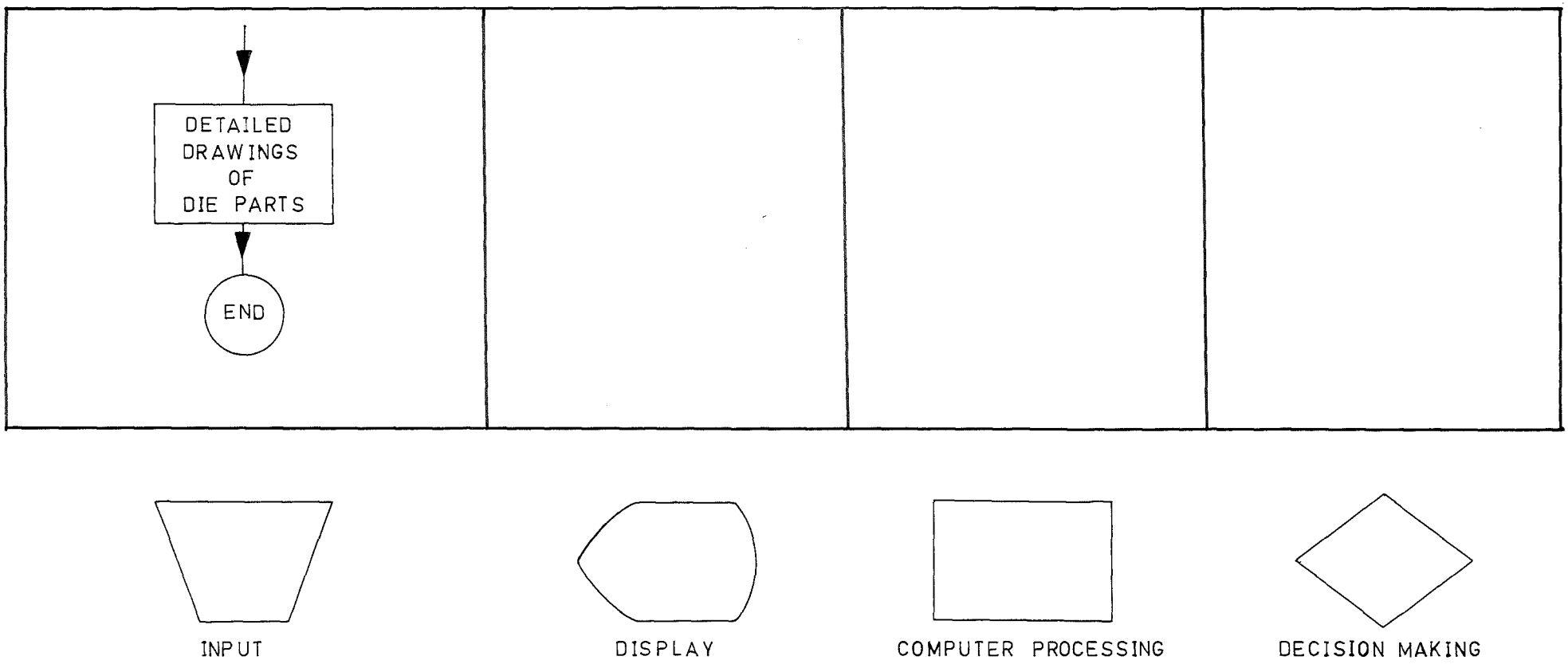


FIG 1.1 THE PROPOSED DESIGN PROCEDURE FOR COMPRESSION AND INJECTION PROCESSES.

of the product, the accuracy required, the material, the economic considerations, and possibly the type of machines available to him. This decision has always been based on experience and common practice, although some attempts have been made to optimize the production method provided sufficient information is obtained.

Here, this decision is best left to the designer who can then feed his decision into the computer. There is a future hope that the computer can run an optimization calculation to make this decision when an optimization process is fully investigated and well established.

(3) Shrinkage Allowance

The product shape now becomes the mould cavity whose overall dimensions have to be slightly bigger than the product itself to allow for the material shrinkage. This is done by a simple multiplication with the shrinkage factor for that material and for the selected production process. The shrinkage factor is a property of the material and the production process and hence it can be calculated internally by the computer.

(4) What machine is to be used? What is the number of cavities?

The two questions are considered simultaneously because they are so closely related to each other.

The optimum number of cavities is dependant upon the rate of production aimed at by the sales office, the machine data: size, clamping force, injection pressure, etc., the cycle time, the cost of manufacture of a single cavity. Unfortunately, many of these quantities are uncertain at this stage such as: the cycle time, the manufacture cost, which makes the optimization process ineffective. Again experience and knowledge of the designer are the determining factors for answering these questions. Hence this decision is

best left to the designer. If he decides to employ multi-cavity mould, he will have to decide on how to arrange these cavities on the die platen. The arrangement of cavities depends on the shape of the product and in general they should be arranged in such a way that all cavities are symmetric about the sprue, for injection moulding, and in such a way as to achieve the shortest possible runner lengths. In other words, this decision requires a pattern-recognition capability which is again best left to the designer. However, these questions are open to further investigation and automation may be possible. A standard die set is automatically selected with the type of the machine.

(5) The Position of Part-line

The position of the part-line is determined by two factors: the artistic appearance of the product and the geometry of the product especially when it has undercuts. From the artistic point of view, it is obvious that the part line should be located at places which are not visible or are not often looked at or do not attract the attention of the people either by sight or by touch. From the geometrical point of view, a well-selected part line can greatly simplify or even eliminate the use of side cores or collapsible cores required by the undercuts.

Since human aesthetic judgement and pattern recognition are involved, this decision making is again best left to the designer.

(6) Which way up is it to be moulded?

The way the cavity is placed depends on the shape and the flow pattern of the material. The material flow of course depends on the product shape and the moulding process and can greatly affect the surface finish of the product. Therefore, this question again requires the designer's decision.

(7) Type of Flash or Type of Sprue, Gate and Runner System

This problem is again dictated by the shape of the product. For example, the sprue and the runner system depends on the size and the shape of the product and must be formed in such a way as to give a good cavity filling rate and good flow pattern around pins, cores, to reduce the tendency of weld-line formations. The type of gate may also be dictated by the method of injection. In general, this question is answered by the designer with his experience and knowledge of die design. Whichever type he decides on, the computer can compute all other necessary dimensions. For example, in compression moulding, once the type of flash has been decided, the powder well depth can be easily computed from the bulk factor of the material. Also the geometry of the power well, the width of land, the flash clearance are easily obtained from standard design codes which have previously been stored in the computer. Hence this process involves a search in the computer storage to access the right data. The same process applies to injection moulding where the geometries of the sprue, the gate and the runner are standardised.

(8) Are there any undercuts that require side cores, collapsible cores or inserts?

This question is closely related to question 5. However, the designer has to decide in the first stage, which details of the product can better be left and completed after moulding as a finishing process, e.g. some holes may be easier to drill after moulding than to be moulded. This decision depending on the product geometry can greatly simplify the die. Also with experience and knowledge, he can design simple cores to produce the desired undercuts in close relation to the position of the part line.

Again, this decision, involving pattern recognition and human judgement, is best left to the designer.

(9) Method of Ejection

The method of ejection, manual, semi-automatic or fully automatic, one stage or multi stage is a decision dictated by the geometry of the product and economic considerations. The designer must also decide on the position, the size, and the number of ejector pins. These decisions are made by the designer.

(10) The Ejection Stroke and the Return Pin Length

The ejection stroke is fixed by the machine data and the return pin length can be easily computed by the computer. The size of the return pin and its shape are standardized and stored in the computer.

(11) The Cooling System

This is only necessary for injection moulding processes. The designer must decide on the cooling fluid, usually water, and the cooling passages. The cooling passages must be such that they fit into the available space left in the die blocks and in such a position that equal mould temperatures are achieved at any point. For example, the passage should be nearer to the walls at the gate than at the cavity.

(12) Is the System Workable?

This is the final stage of the design. The designer must check on all details particularly the mechanisms of side and/or collapsible cores and of the ejection process. Dimension checking is unnecessary.

This step at the moment is left to the designer's ability to imagine the system in operation. However, with further investigation on the possibility of generating motion pictures, the working mechanisms could be checked by the computer.

If the designer finds any error or wishes to make any modification

to his system, he can go back as far as step 4 and he can do this as many times as he wishes, thus generating many alternatives. Otherwise, the general die assembly is completed.

(13) Extraction of Individual Die Components

Individual die components are now ready for extraction from the general assembly. For example, die blocks must have holes to allow pins to pass through or to hold inserts. Each hole must also be allowed a suitable clearance depending on its type.

Although the extraction process is straight forward to the designer, it does need some degree of understanding of the shapes of the individual parts and their intended use. This step could be done automatically by the computer but the process would be slow because it involves testing one object with every other object to see if they intersect or touch. When two intersecting or touching objects are found, it has to decide further whether to add, subtract or ignore them. As a compromise, it has been left to the designer to decide which objects are to be added or subtracted while the computer does the actual addition and subtraction.

(14) Detailed Drawings of Die Components

In principle, the computer can produce detailed engineering drawings of die components since it has their shape descriptions. Even so, it is not a simple process and a thorough study of computerized dimensioning of drawings is needed. On the other hand, it can also be done interactively with the user who provides the necessary information, e.g. the positions and the datum for the dimensions, possibly with the use of a light pen.

Having gone through the entire design procedure, the major question is how an object is described to a computer. Objects either existing in nature or created by man are in a vast variety of forms varying from the most simple to those of extreme complexity. It is therefore very difficult to implement a computer graphics system with the capability of describing all objects. Very often, this is not desired either because a certain application only requires objects of a certain class of geometry. Examination of typical dies show that they are usually made up from shapes bounded by flat, cylindrical and conical surfaces. Various methods of description are discussed in Chapter 2 and the selected one, the block building method, is described. It should be emphasized here that there is a coincidence between the idea of this technique with that of I.C. Braid's "Building with Volumes" [1]. This work started on 4th March 1974 as registered with the University of Canterbury while the work of Braid in University of Cambridge, England, was first published on April 1975 ("The Synthesis of Solids Bounded by Many Faces", ACM, No. 4, Vol. 18). During this period, the idea of the technique was developed independently. Since then, Braid's work has been extensively referred to. However, it should be noted that Braid's work was limited in the class of object shapes used and it differed in many details.

1.3 The Expected Advantages

This computer implementation can be expected to bring many advantages:

(1) Shorter lead time: it may take a few days to complete a design compared with weeks by conventional methods.

(2) The die designer is freed from the tedious, non-creative job of producing detailed drawings of individual components.

(3) Self-checking so far as arithmetic is concerned.

(4) Since co-ordinates of every point are held or may be computed,

the output could be in the form of numerically controlled tapes for NC machines, and hence automation in manufacture is possible.

(5) For custom moulding work, it enables a design to be finished in a short time, which in turn should enable tool costs to be estimated with greater accuracy and perhaps eventually automatically.

(6) The generation of alternative designs with different runner systems and numbers of cavities, etc. could be achieved in a relatively short time and should yield a more accurate optimum die design.

(7) The possibility of moving pictures on graphics screen could provide an efficient way of checking the cams and the ejection mechanisms.

CHAPTER 2

THE DESCRIPTION OF A GENERAL OBJECT IN THREE DIMENSIONS

2.1 General Object Description

An investigation into the work that has been done by computer graphics specialists shows that an object, in general, is defined in a tree-type structure in terms of its vertices, its lines, and its surfaces.

(1) Vertices

Vertices are numbered and lie in the bottom level of the tree structure. Their three-dimensional co-ordinates are stored in a list.

(2) Edges

An edge is a finite line segment joining two vertices together. Edges are numbered and stored in another list. Each edge contains information to show whether it is straight or curved and pointers to show its starting vertex and its end vertex.

(3) Faces

Faces are finite areas on a surface bounded by edges. Faces are stored in a list, which contains information to show all the edges that constitute the boundary of each face and the edges constituting hole(s) if any. It also contains a pointer to show whether the face is flat or curved.

Flat faces are found to be satisfactorily described by the implicit equation in the form:

$$Ax + By + Cz + D = 0.$$

Curved faces are found to be best described in its parametric form. Simple curved surfaces such as cylindrical and conical, are easily described by one parametric equation as in [1] and shown in Appendix E. Other general curved surfaces require

piece-wise approximations. Each surface is considered to consist of many small patches joined together at their boundaries under certain constraints: Coon's patches [2], Bézier and Ferguson patches [10] and other types of patches [7]. In other applications, a curved surface can be defined as a sequence of cross sections [3, 8], each of which is a plane curve perpendicular to a common spine which is a curve itself in three dimensional space. The patch method is suitable for any general curved surfaces that cannot be represented by simple parametrical or analytical equations while the cross section method is suitable for closed, curved surfaces possessing some degree of symmetry. All piecewise representations require a tremendous amount of storage space and are very slow to process. The choice of a particular method thus depends very much on the geometry of the shapes.

Although objects can take a wide range of shapes varying from the simplest one to the most complex, a computer graphics system does not have to have the capability of describing objects of all shapes because for each application, only a certain class of shapes is required. Examination of typical plastics dies shows that they are usually made of shapes bounded by flat, cylindrical and conical surfaces although there are dies that have far more complicated shapes. Since this work was to test and prove the feasibility of an idea, it is sufficient to restrict object shapes to have flat, cylindrical and conical surfaces. The use of piece-wise representation therefore becomes unnecessarily general because these surfaces can be easily represented in parametric, rational polynominal forms.

In this class of objects, Fig. A-1 (Appendix A) shows the full object definition of a box, which is required to be stored in a computer. This data structure tells the computer that the object

has 8 vertices at the given co-ordinates, 12 straight edges joining between them and also forming 6 flat faces whose equations are given. This is a very well defined data structure as far as the computer is concerned. However, if the user has to prepare and enter separately a complete data structure such as this for every object, it is not so good. He would be very prone to making mistakes, in addition his data structures might not be consistent and uniform. It is much simpler for him to define a box by specifying the co-ordinates of its centroid, say, in space and three dimensions along its sides: length, width, height. He would not wish to specify the co-ordinates of all the vertices, line pointers, surface pointers, and surface equations. The preparation work would be even more tedious for a T-bar e.g. as shown in Fig. 2.1. even though this is still a very simple shape. The solution to this problem lies in the block-building technique. Using this technique, an object is built by putting unit blocks of standard shapes together, in the same manner as houses are built from rectangular bricks. The full definition of each unit block has to be prepared only once and stored. When the user wishes to define an object, say a box, he just has to input the minimal amount of data: the co-ordinates of the centroid and three dimensions along its sides which are parallel to the co-ordinate axes. On receiving this information, the computer generates the required box by modifying the previously established one. That is, the unit box is scaled up to the required size and translated to the required position. The scaling and translation operations are simple and can be easily done by the computer since they only involve matrix multiplications. Only the co-ordinates of the vertices and face equation co-efficients are modified while other features like line pointers, face pointers, etc.

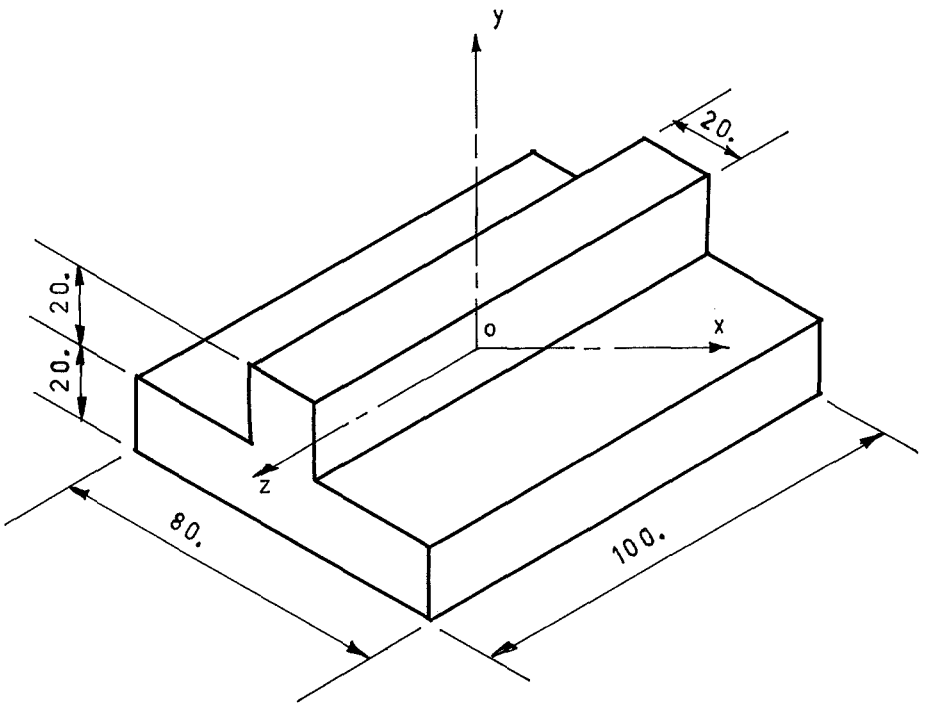


FIG. 2.1. A T-BAR

remain unchanged. Full details of the technique are discussed in the following section. There is a coincidence between the concept of this technique and that of "Building with Volumes" developed by Braid [1]. When Braid first published his technique in April 1975, the idea and the technique had been developed independently at this university and the process was well under way. However, since that time, Braid's work has had a great influence on this project although his method has been modified and extended to the larger class of shapes needed for this application.

2.2 The Block - Building Technique

In this technique a menu of 6 unit objects, i.e. a box, a cylinder, a wedge, a fillet, a truncated cone and a truncated pyramid is set up. Each unit object has its centroid, or reference point, at the origin of the co-ordinate system and has some convenient dimensions. The data structure for each unit is prepared (see Appendix A) and stored. Chapter 3 discusses them in full detail. The building block, called a primitive object, is any one of the unit objects after it has been modified to its required dimensions, position and orientation in space. The algorithm to transform a unit object into a primitive object is in any elementary geometry text book. It involves only the co-ordinates of vertices and the face equations. Face-edge pointers and edge-vertex pointers remain unchanged. The user can define as many primitive objects as required. A primitive object can also be made positive or negative. A positive object is considered as a solid object while a negative object is considered as an empty object. Two or more of these primitive objects can then be put together to build a new object in a well-defined manner and their data structures are combined accordingly. The building routines are merging and intersection. It should be noted that Braid's

work did not cover conical surfaces and one of the two objects in the intersection routine must be a primitive box or a primitive cylinder.

2.2.1 Merging

Merging adds or subtracts two objects which do not intersect each other and which have at least one pair of flat, coplanar faces.

When two positive objects which lie outside each other are merged, they are simply joined together at their coplanar faces (Fig. 2.2(a)). Symbolically, the resultant object C is the union of the original objects A and B:

$$C = A \cup B$$

When a negative object which lies inside another positive object and is merged into it, the result is to remove the part of the material having the shape of the negative object from the positive object (Fig. 2.2(b)).

It should be noted that each pair of coplanar faces (coplanar faces must also be of opposite sense) requires one pass of the merging algorithm. Two objects, therefore, may require more than one merging to be fully combined.

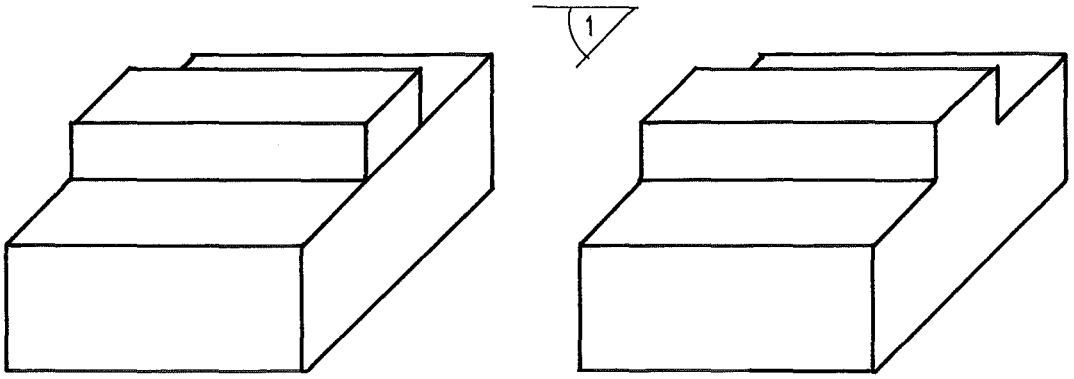
The characteristics of merging are:

1. New vertices can be created.
2. No new edges can be created. Any edge of the resultant object is an edge or part of an edge of the original objects.
3. When an original face is deleted, it is always replaced by a new face or new faces lying on the same plane.

The complete merging algorithm is given in Chapter 4.

2.2.2 Intersection

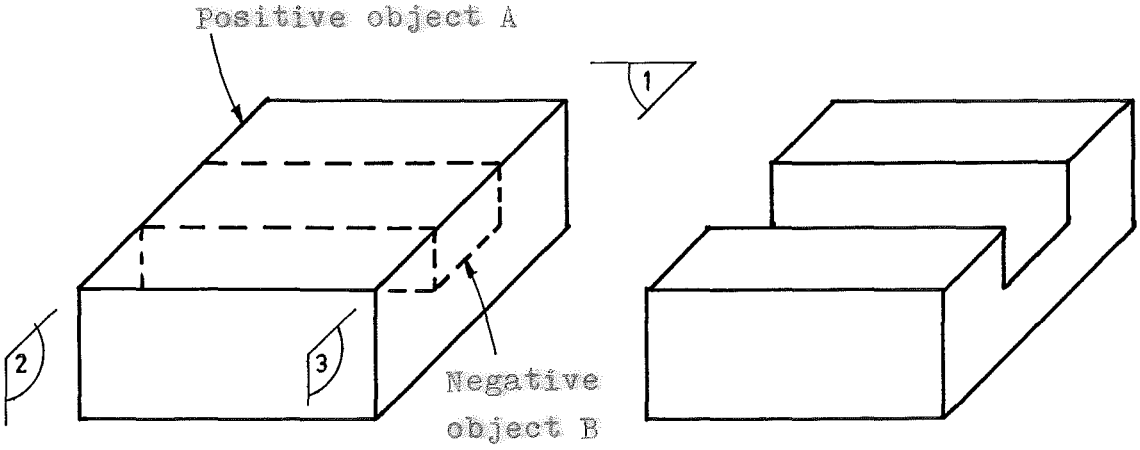
Intersection also adds or subtracts two objects but the two objects must intersect and do not necessarily have to have any pair of flat coplanar faces.



Two positive objects.
One pair of coplanar faces.
Before merging.

After one merging.

(a)

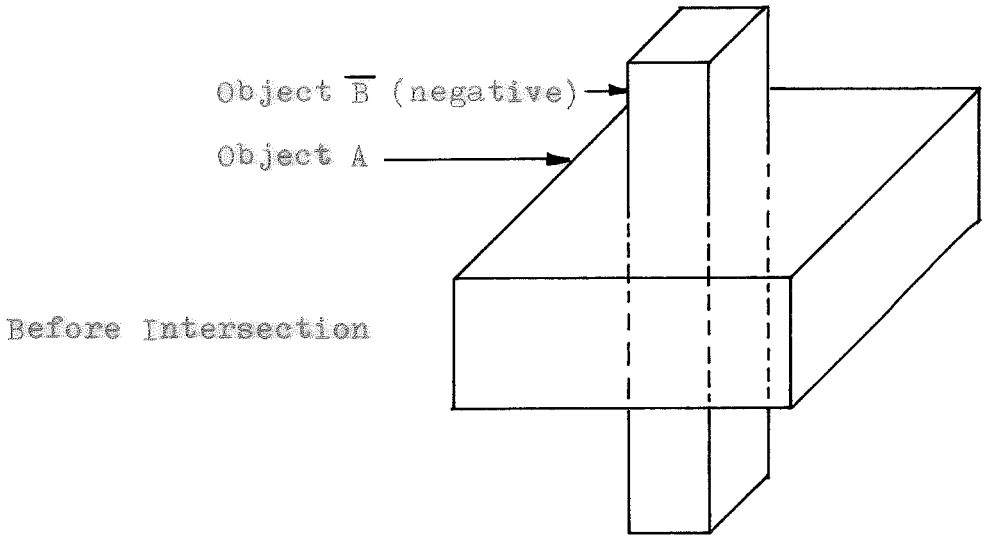


Three pairs of coplanar faces.
Before merging.

After three
successive
mergings.

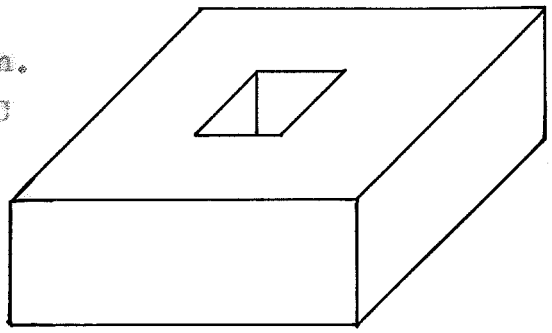
(b)

FIG.2.2. Merging.



Before Intersection

(a) After Intersection.
Resultant object C
$C = A \cap \overline{B}$



(b) Resultant object C
where :
$C = \overline{(\overline{A} \cap \overline{B})}$

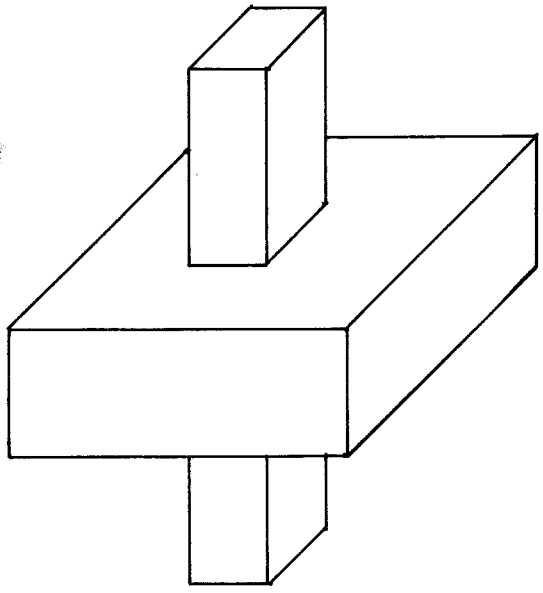


FIG. 2.3. Intersection.

The basic intersection algorithm finds the object C resulting from the intersection of a positive object A and a negative object $\bar{B}$. Object C has all the material inside object A and outside object B (Fig. 2.3(a)). Symbolically, object C is the intersection of object A and $\bar{B}$:

$$C = A \cap \bar{B}$$

However, intersection algorithm can also be used to add two positive intersecting objects A and B i.e. to find their union. This is done by first negating A and B, then intersecting $\bar{A}$ and $\bar{B}$ and finally negating the resultant object (Fig. 2.3(b)). This is true because the resultant object from intersection $\bar{A}$ and $\bar{B}$ is $(\bar{A} \cap \bar{B})$ and the negative object of this is $\overline{(\bar{A} \cap \bar{B})}$, which is simply AUB.

Two intersecting objects only require one pass of intersection algorithm no matter how many pairs of coplanar faces they have.

The characteristics of the intersection algorithm are:

1 - New vertices can be created.

2 - New edges can be created.

3 - An original face can be completely deleted without being replaced by any new faces.

In Braid's intersection algorithm, the negative object $\bar{B}$ must be not only a primitive object but also a primitive box or a primitive cylinder. This restriction was imposed to simplify the algorithm. Here this restriction is lifted to allow $\bar{B}$ to be any object.

The complete intersection algorithm is given in Chapter 5.

2.2.3 Advantages

The block-building method has many advantages:

(a) The user need not be skilful or well-trained because data input is easy and simple.

(b) It eliminates the user's possible mistakes in preparing input data.

(c) It saves core storage because the exact size of the data structure

of a primitive object is known.

To illustrate these points, let us generate the T-bar shown in Fig. 2.1.

1. Define 2 primitive boxes by calling the box defining sub-routine (see next Chapter) then enter:

   * box A: dimension along x, y, z axes:

     20., 20., 100. units and reference point (centroid) at

     0., 10., 0. units

   * box B: dimensions: 80., 20., 100.

     reference point at: 0., -10., 0.

2. Call the merging routine to merge A and B at their common faces. The T bar will be created at the end of the routine.

CHAPTER 3

THE DESCRIPTION OF PRIMITIVE OBJECTS

Primitive objects are defined through "unit" objects. Unit objects are objects formed with fixed, convenient dimensions and with their centres or reference point at the origin (0., 0., 0.). Their dimensions are not of unit length as the name may suggest.

Fully-detailed descriptions of unit objects are given in Appendix A, their data structure being outlined in Appendix B. All unit objects are stored as positive objects. The data structures of positive objects are formed according to the following conventions:

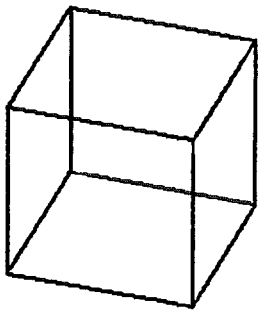
1. The normal vector of a face is a vector which is perpendicular to the face and points in the direction from the inside of the object to the outside.
2. The edges (and their directions) which form a closed loop (e.g. the perimeter) of a face are stored in a strict order. The convention is that the perimeter of a face is traced in anti-clockwise direction when it is looked at against the direction of the normal vector. It follows that all loops which are holes on a face are traced in the opposite direction.

When the user wishes to define a primitive object, he is required to enter:

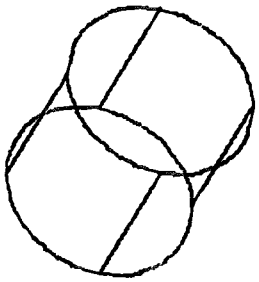
1. The name of the object in three-letter words shown in Fig. 3.1.
2. The dimensions of the object.
3. The reference point of the object.
4. The orientation of its axis.

The general procedure of the program would be:

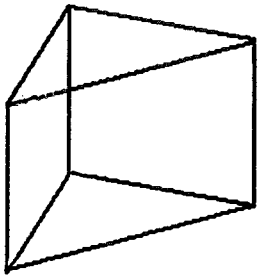
1. To get the required defining routine and call out the data structure of the corresponding unit object.
2. Form a 4 x 4 matrix S, the scaling matrix, to scale the object to the



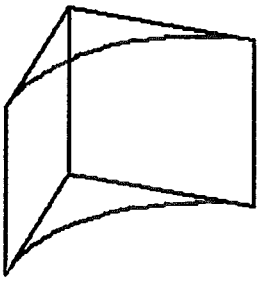
BOX



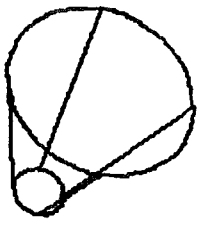
CYL



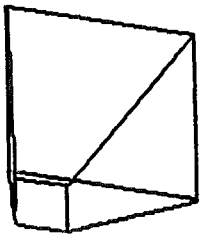
WED



FIL



CON



PYR

FIG. 3.1. Six primitive objects.

required dimensions.

3. Form a rotation matrix R, also 4 x 4, to rotate the pre-determined axis of the object to its new orientation.

4. Form a translational matrix [TL] to move the object in the direction of vector T. The distance moved is given by the length of vector T. Vector T is the vector which starts at the reference point of the unit object and ends at that of the primitive object.

5. Form a total transformation matrix [TR] to transform the unit object to the primitive object by matrix concatenation where:

$$[TR] = SR[TL]$$

6. Calculate the new vertices' coordinates and surface equations of the primitive object from those of the unit object and [TR].

The data structure of the primitive object would be stored and could be recalled for further transformation if necessary. The procedure is summed up in the flow chart of Fig. 3.2. As many primitive objects of any kind can be defined as required.

Every primitive object defined in this way is a positive object, in other words, a solid object. Objects can also be made negative by inputting a NEG (for negate) command. Negative objects are hollow objects of infinitesimally thin walls which can be merged with a positive object effectively removing the part of material of their shapes from the positive object.

When a positive object is negated, the normal vectors of all of its faces are reversed in direction. In addition, the directions of the edges which form a closed loop and their order of appearance in the face-edge list are also reversed. When an object (positive or negative) is negated twice, its data structure changed back to what it was before.

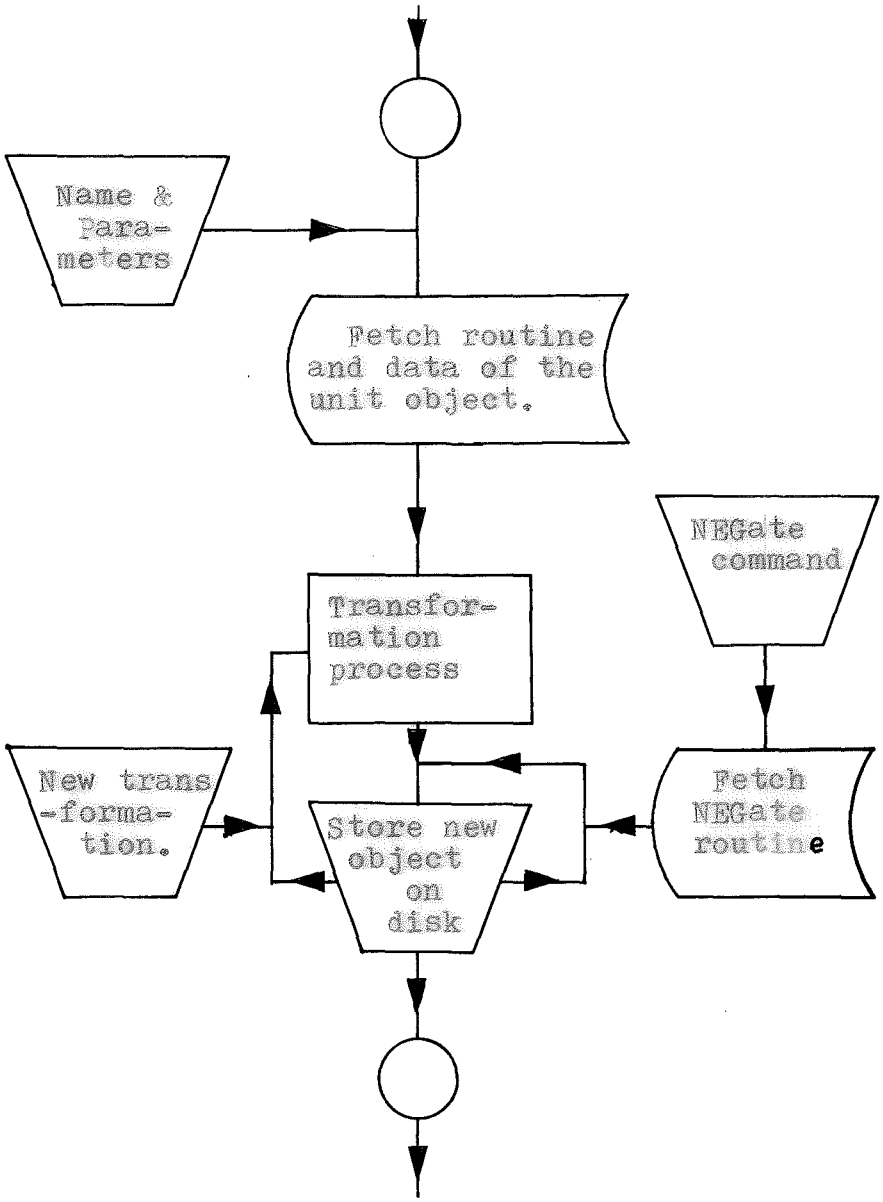


FIG.3.2. The flow chart of generating primitive objects.

CHAPTER 4

MERGING

Merging and Intersection are the two algorithms which were first developed and described by Braid in his work [1]. His descriptions are found to be ambiguous in some stages, especially in the handling of a pair of colinear, overlapping edges in his merging algorithm. Other descriptions are given here and in Chapter 5. Hopefully they will clarify the ambiguities but it is pointed out that the algorithms are basically the same as those of Braid.

The merging algorithm combines two non-intersecting objects into another object at their flat, coplanar faces. Physically it is like sticking two objects together at the common flat faces. When one object is negative, the effect is to remove the volume of material having the shape of the negative object from the positive object.

The algorithm first finds out how many pairs of flat, coplanar faces the objects have and records them. For each pair of coplanar faces, the edges lying on the common plane are examined in pairs. An edge pair consists of an edge or an edge piece from one object and another edge or edge piece from the other object. All intersection points between edges are recorded and edges are broken up into pieces at these points. New vertices may be created. Then the edges or the edge pieces are joined together again to form a new face or new faces of the resulting object. Special attention is given to the case where two edges are colinear and overlapping: one edge may appear to be like a crack in the final object and has to be removed. The resulting object may have new vertices and new faces but the edges are always the edges or pieces of the edges of the original objects.

There are four stages in merging. They are executed in succession and may be repeated for each pair of coplanar faces. Merging is carried

out when the user runs the batch file TGMERG and enters:

record number 2 (object 2), record number 1 (object 1)

which means "merge object 2 into object 1".

The reader should consult Appendix B for the data structure.

4.1 Stage One

The data structures of the two objects are read out from their files and then lumped into one single data structure. Each feature in the structure of object 2 is copied right after the corresponding feature of object 1 to make that feature of the combined structure of the resulting object.

1. Vertices of object 1 are numbered from 1 to NP1, those of object 2 from 1 to NP2. The combined structure will have vertices numbered from 1 to NP where NP = NP1 + NP2. Vertices from 1 to NP1 are exactly those of object 1 and vertices from (NP1 + 1) to NP are exactly those of object 2.

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_{NP1} & y_{NP1} & z_{NP1} \end{bmatrix}$$

Coordinates of vertices of object 1

$$\begin{bmatrix} x'_1 & y'_1 & z'_1 \\ x'_2 & y'_2 & z'_2 \\ \vdots & \vdots & \vdots \\ x'_{NP2} & y'_{NP2} & z'_{NP2} \end{bmatrix}$$

Coordinates of vertices of object 2

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_{NP1} & y_{NP1} & z_{NP1} \\ x'_1 & y'_1 & z'_1 \\ \vdots & \vdots & \vdots \\ x'_{NP2} & y'_{NP2} & z'_{NP2} \end{bmatrix}$$

Coordinates of vertices of the combined structure

Hence the matrix holding the coordinates of the vertices will be formed as shown.

2. The edges of the combined structure are numbered similarly from 1 to NSE, say for straight edges, where NSE = NSE1 and NSE2. Suffix 1 refers to object 1 and suffix 2 refers to object 2.

$$\begin{bmatrix} V_{11} & V_{12} & 0 & 0 \\ V_{21} & V_{22} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ V_{(NSE1)1} & V_{(NSE2)2} & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} V'_{11} & V'_{12} & 0 & 0 \\ V'_{21} & V'_{22} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ V'_{(NSE2)1} & V'_{(NSE2)2} & 0 & 0 \end{bmatrix}$$

Straight edge list of object 1 — Straight edge list of object 2

$$\begin{array}{ll} \text{edge } 1 & \begin{bmatrix} V_{11} & V_{12} & 0 & 0 \\ V_{21} & V_{22} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ V_{(NSE1)1} & V_{(NSE1)2} & \vdots & \vdots \\ V''_{11} & V''_{12} & \vdots & \vdots \\ V''_{21} & V''_{22} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ V''_{(NSE2)1} & V''_{(NSE2)2} & 0 & 0 \end{bmatrix} \\ 2 & \\ & \\ & \\ & \\ \text{NSE1} & \\ \text{NSE1} + 1 & \\ \text{NSE1} + 2 & \\ & \\ & \\ & \\ \text{NSE1} + \text{NSE2} & \end{array}$$

Straight edge list of the combined object.

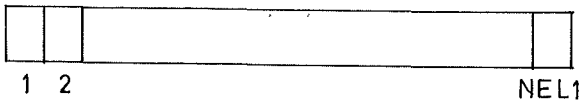
In this instance, however, only edges from 1 to NSE1 have exactly the same end vertices as those of object 1. The vertices of edges from (NSE1 + 1) to NSE are those of the corresponding edges in the list of object 2 plus NP1

i.e. $V''_{11} = V'_{11} + NP1$, etc.

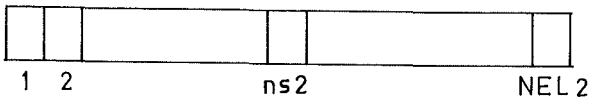
The reason is that the vertices of object 2 are no longer numbered from 1 to NP2 but from (NP1 + 1) to (NP1 + NP2) as discussed in step 1.

If there are any curved edges the curved edge list is constructed in the same manner bearing in mind that all numbers referring to the L group, PL group, curved face number, of edges from (NCE1 + 1) to (NCE1 + NCE2) must be appropriately incremented because the L group, PL group (if any) and curved faces are all merged into those of object 1. 3. Also faces of the combined structure are numbered from 1 to (NF1 + NF2). Flat faces are from 1 to (NFF1 + NFF2) and curved faces from 1 to [(NF1 - NFF1) + (NF2 - NFF2)].

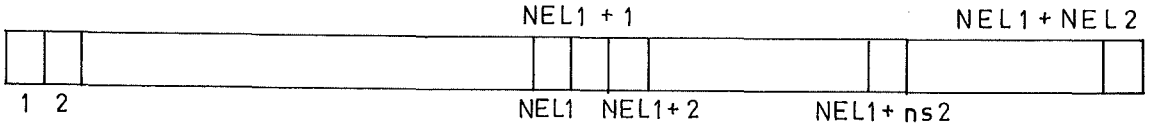
The arrays holding flat face equation coefficients and parametric equation coefficients are constructed in a similar manner to array P which holds coordinates of vertices. And so is the edge list EL for the faces. Note that the edge numbers of all edges lying on faces from (NF1 +1) to (NF1 + NF2) must be appropriately incremented. Appropriate increments must also be made when forming the face status array FS.



Edge list EL1 for faces of object 1



Edge list EL2 for faces of object 2



Edge list EL for the combined structure

For example, in object 2, the starting address of edge list of a face N2 in EL2 is ns2. In the combined structure, face N2 becomes face (NF1 + N2) and its starting address in list EL is now (NE1 + ns2).

In effect, a compact data structure for one single object is now formed although it still describes two separate entities. Note that in forming the new structure, all features belonging to object 1 are completely unchanged. Only the vertex numbers, edge numbers, face numbers, etc. of those features which belong to object 2 are incremented.

The procedure continues:

4. Normalise the coefficients of all flat face equations, i.e. normalise A, B, C, D of equation

$$Ax + By + Cz + D = 0 \qquad (4.1)$$

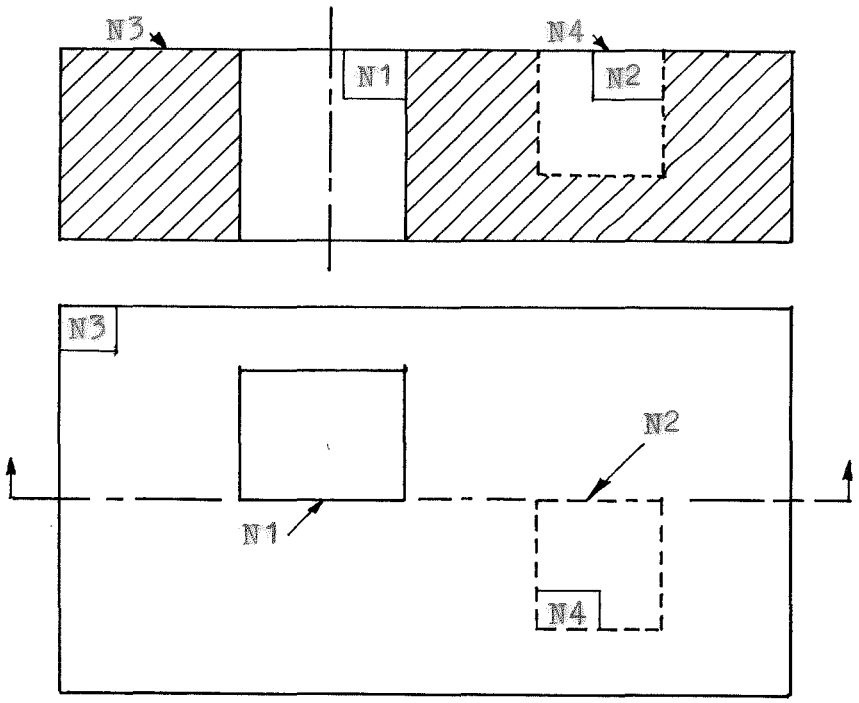
such that $A^2 + B^2 + C^2 = 1$.

Note that A, B, C are thus the direction cosines of the unit normal of the face. The unit normal is outward for a positive face, inward for a negative face.

5. Examine all flat faces in pairs. Two flat faces can be merged if they are of opposite sense, i.e. if

$$(A_1 + A_2)^2 + (B_1 + B_2)^2 + (C_1 + C_2)^2 + (D_1 + D_2)^2 \neq 0. \qquad (4.2)$$

If such a pair cannot be found, print error messages and exit from the routine i.e. return to ready status. When test (4.2) is satisfied for a particular pair of flat faces, the two faces still need another test to see if they can in fact be merged. An example is shown in Fig. 4.1.



——— : Object 1

– – – – : Object 2

FIG.4.1.

Here object 1 and object 2 are meant to be merged at faces N3 and N4. However, each face appears to have no boundary at all when it is represented by an implicit equation of the form (4.1). Therefore equations of faces N1 and N2 will still pass test (4.2) although they could not be merged.

Such a pair of faces can only be merged if their outer perimeters are not disjoint. The test is done in a routine called COMBO. All suitable pairs of flat, coplanar faces are found and recorded.

When two objects are found capable of being merged, the algorithm proceeds to stage 2. All stages 2, 3 and 4 are repeated for each pair of coplanar faces, because they handle only one pair at a time.

Comments:

1. In forming a combined data structure for the new object, sufficient empty spaces should be allowed in each list to store additional features which will arise in later stages of merging e.g. the creation of new vertices, new faces or pieces of a split edge. This can only be guessed at because there are no general rules to predict how many more features will be formed. If at any stage

a particular list is overfilled, an error message will be printed and control is returned to ready status. The user then has to allow more space in that particular list and re-run the routine.

2. In Braid's work, merging still continues although two outer perimeters are disjoint until it is found out in stage 3 of merging and control exits from there. In the present work to avoid unnecessary processing, another step is implemented to eliminate this situation right from the start.

4.2 Stage Two

In stage 2, each edge of face 1 is tested for intersection with each edge of face 2. Whenever an intersection is found, one edge or two edges will be split into pieces and the test continues, but this time whenever a split edge comes into consideration its pieces are examined instead. The general routine of stage 2 is given here while sections 4.2.1 and 4.2.3 give more details of how to handle intersecting and overlapping edges.

1. Find starting addresses of face 1 and face 2 and the numbers of edges belonging to face 1 and face 2.

2. Get the next edge on face 1. Note if it is straight, cylindrical or conical. Let IL1 be a temporary list containing the current edge (and its pieces) which belong to face 1. Clear list IL1 and put this edge at the beginning of the list. Set ILP1, the pointer of this list, to 1.

   If the current merging is not the first emerging, the current edge might have been split. If so, get its pieces and put them into IL1. Again set ILP1 to 1.

3. Get the next edge E1 from IL1 list as indicated by pointer ILP1. The edge from this list might be an original edge or a piece of a split edge. If there are no more edges or pieces in IL1 list, go to step 22.

4. Check on the edge list EVS (straight) or EVC (curved) to see if edge E1 has been replaced or deleted. If E1 has been deleted, go to step 21. If E1 has been replaced by another edge, set the replacing edge as the current edge E1 and re-check E1 by going back to the start of step 4. If E1 has not been deleted or replaced by another edge, continue on step 5.

5. Get the next edge on face 2. Note if it is straight, cylindrical or conical. Call this edge E2.

6. Check on EVC or EVS list to see if:

   (a) E2 has been deleted. If it has, go to step 20

   (b) E2 has been replaced by another edge. If it has, call the replacing edge E2 and go back to the beginning of step 6.

   (c) E2 has not been deleted or replaced by any other edge, go to step 7.

7. Let IL2 be a temporary list containing the current edge of face 2 or its pieces under consideration. Clear list IL2 and put E2 at the beginning of list IL2. Set ILP2, the pointer of this list, to 1. If edge E2 has been split (if indicated by EVS or EVC list), call a routine to replace edge E2 with all of its pieces in list IL2.

8. Call the next edge in list IL2, as indicated by the pointer ILP2, edge E2. If there are no more edges in list IL2, go to step 20.

9. If E2 has not been deleted or replaced by any other edge, go to step 10.

   If E2 has been deleted, go to step 12.

   If E2 has been replaced by another edge, call the replacing edge E2 and go back to the beginning of step 9.

10. Check to see if E1 is exactly the same as E2. If it is, go to step 12.

11. Examine edges E1 and E2 for any intersections.

    If there is no intersection, go to step 12.

    If there is one intersection, go to step 13.

    If there are two intersections, go to step 14.

    If E1 and E2 are overlapping, go to step 15.

12. Increase ILP2 by 1 and go to step 8.

13. Call a subroutine to handle the one-intersection case. See Section 4.2.1 for more details. Briefly, the following actions may be taken:

    - no action at all may be taken

    - an old vertex may be deleted

    - a new vertex may be created

    - edge E1 might be bisected. In this case, list IL1 is pushed down one space at ILP1 to make room for the replacement of E1 by its 2 pieces in this list.

    The pointer ILP1 is left as it was and therefore it is now pointing at the first piece of edge E1.

    - edge E2 might be bisected. In this case, list IL2 and its pointer ILP2 are treated in a similar manner to the case in which edge E1 is bisected.

    Then go to step 16.

14. Call a subroutine to handle the two-intersection case. See sections 4.2.2 for more details. Briefly, the following actions may be taken:

    - no action at all may be taken

    - an old vertex or 2 old vertices may be deleted

    - a new vertex or 2 new vertices may be created

    - one edge or two edges may be split (bi-sected or tri-sected) and their edge lists IL1 and IL2 are updated correspondingly, similar to step 13.

Then go to step 16.

15. Call a subroutine to handle the case of overlapping edges. See sections 4.2.3 for more details. This is the most challenging step in merging because it opens to so many different actions that may be taken:
    - no action at all may be taken
    - one old vertex may be deleted
    - one edge might be bisected and the other deleted. The corresponding edge lists must be updated in this case
    - two edges might be bisected and the common piece deleted
    - one edge might be trisected and the other edge deleted
    - a routine might be called to remove a "crack". Then go to 16.
16. If edge E2 has been deleted or replaced by another edge, go to step 3.
17. If edge E2 has not been split, increase ILP2 pointer by 1. If it has been split, ILP2 remains as it was.
18. If edge E1 has been deleted or replaced by another edge, go to step 4.
19. If edge E1 has not been split, go to step 8. If edge E1 has been split, go to step 3.
20. If there are any more original edges on face 2, go to step 5. If there are no more original edges in face 2, go to step 21.
21. Increase ILP1 pointer by 1 and go to 3.
22. If there are any more original edges on face 1 to be considered, go to step 2. Otherwise, every possible pair of edges, one from face 1 and the other from face 2, have been examined for intersection and stage 2 of merging ends.

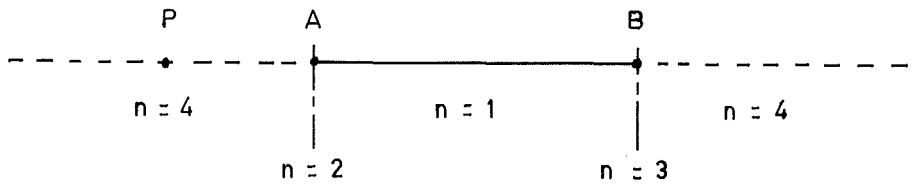
In this routine, all edges of face 2 are repeatedly tested against each edge of face 1 for intersection. Whenever an edge in face 1 is split, it is replaced by its pieces and then the first piece of this edge

is tested again against all edges of face 2, starting from the first edge. In this way, an edge or part of an edge in face 1 may be tested more than once against an edge in face 2, which is an unnecessary step. Strictly speaking, every edge in face 2 which has been tested with a particular edge in face 1, should be recorded so that it will not be scanned again when part of that edge in face 1 is encountered later on. Of course, the recording of such data would require more storage and hence is not used.

The challenge in this stage is not finding the intersections but how to record them, handle them and update the data structure. The following sections will discuss in details the cases where there is one intersection or two intersections and where the edges overlap.

4.2.1 One Intersection Point

Let the end vertices of edge 1 and edge 2 be A1, B1 and A2, B2 respectively. Appendix D shows how the intersection of 2 straight edges is described in terms of a quintuple $(n_1, n_2, n_3, n_4, n_5)$. In this case, n_5 being the number of intersections must be 1 and only numbers n_1 and n_2 are considered to describe the position of the point of intersection PI with respect to edges 1 and 2 respectively. The numbers n_1 and n_2 are briefly defined as follows:



Consider a point P and an edge AB going from A to B. Number n describes the position of P on AB. If:

n = 1: point P is inside edge AB.

n = 2: point P coincides with vertex A.

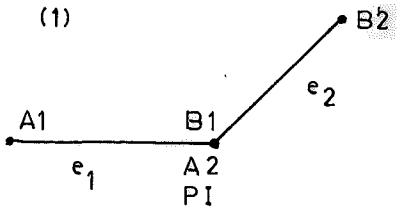
n = 3: point P coincides with vertex B.

n = 4: point P is outside edge AB.

Fig. 4.1 summarises all possible actions which may be taken in this case. The handling of the edges in this case follow the following routine:

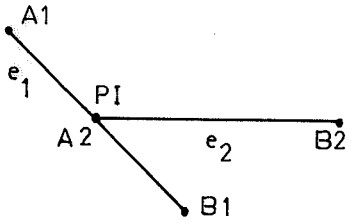
Actions:

(1)



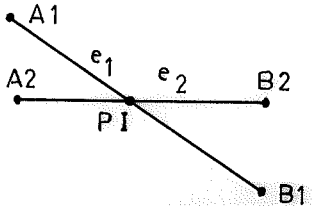
Two vertices coincide. If there are actually 2 vertices existing at this point, delete one of them.

(2)



The intersection point is at one vertex of one edge and bisects the other edge (in this case edge e_1).

(3)



One separate intersection. Create a new vertex at this intersection point and bisect two edges.

Fig. 4.1 All possible actions taken in the case of one intersection between straight-straight edges. Actions are the same when curved edges are involved.

1. Set FV1 = 0.

   Set FV2 = 0.

2. If n_1 = 2, set FV1 = A1.

   If n_1 = 3, set FV1 = B1.

   If n_2 = 2, set FV2 = A2.

   If n_2 = 3, set FV2 = B2.

3. If both FV1 and FV2 are non-zero, the intersection point coincides with vertex FV1 of edge 1 and with vertex FV2 of edge 2. In this case go to step 8. This is typically shown in Fig. 4.1.(1).

4. If either FV1 or FV2 is non-zero, the intersection point coincides

with only 1 vertex of one edge as typically shown in Fig. 4.1.(2). In this case, go to step 6.

5. At this step, both FV1 and FV2 are all zero, which means the intersection point is inside both edges as shown in Fig. 4.1.(3). In this case, create a new vertex, numbered (NP + 1), at the intersection point. Then replace NP by (NP + 1). Also enter this vertex into the special vertex list ISP. A special vertex is a vertex that lies on both face 1 and face 2. Set FV1 = FV2 = NP.

6. If $n_1 \neq 1$, go to step 7.
Otherwise, n_1 must be 1. In this case, bisect edge 1 at FV2, to make 2 new edges (which are actually pieces of edge 1):
- edge (NSE + 1) or (NCE + 1) goes from A1 to FV2
- edge (NSE + 2) or (NCE + 2) goes from FV2 to B1
- if FV2 has not been in the ISP list, put it in that list.
Also list IL1 is pushed down 1 space at pointer ILP1 and enter edge (NSE + 1) or (NCE + 1) and (NSE + 2) or (NCE + 2) at addresses (ILP1) and (ILP1 + 1) respectively. Leave the pointer as it is. Replace NSE (or NCE) by (NSE + 2)(or (NCE + 2)).

7. If $n_2 \neq 1$, exit from the routine. Otherwise n_2 must be 1. In this case, edge 2 is bisected at vertex FV1 in a similar manner to step 6. List IL2 is also updated in the same way. Then exit from the routine.

8. If FV1 is the same as FV2, exit from the routine.
Otherwise let:
NDS = FV1 or FV2, whichever vertex is smaller
NDG = FV1 or FV2, whichever vertex is bigger.
Then:
- delete the coordinates of vertex NDG from the list P and update list P. In this way, all vertices which were previously smaller than NDG are now unchanged whereas all vertices which were previously

bigger than NDG now have their numbers, each reduced by 1.

- check through EVS and EVC list and whenever:

* vertex NDG is met, replace it by NDS.

* a vertex having number greater than NDG, reduce it by 1.

- enter NDS into list ISP if it has not been in the list. Then exit from the routine.

In this routine, an action such as vertex creation or deletion is performed right away and the data structure is updated immediately instead of just recording it on a separate list and operating on it when all new vertices or all deleted vertices have been found because the latter method would have complicated the procedure.

4.2.2 Two Intersection Points

This case may arise between one straight edge and one curved edge or between two curved edges.

In this case n_5 in the quintuple $(n_1, n_2, n_3, n_4, n_5)$ must be equal 2; and (n_1, n_2) describe the relative position of the first intersection point PI1 with respect to edges 1 and 2 while (n_3, n_4) describe the relative position of the second intersection point PI2.

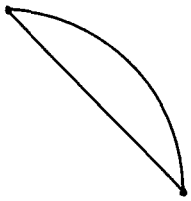
Fig. 4.2 summarises all possible actions which may be taken in this case. Although the illustration gives the case of one straight and one curved edge, it applies to the case of 2 curved edges as well.

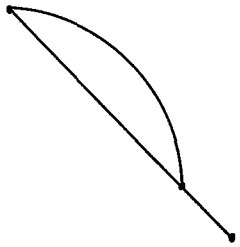
However, it is very useful to note that each intersection point in this case can be treated separately, that is as though the other intersection point did not exist and hence the routine for handling one intersection, given in the previous section, can be employed.

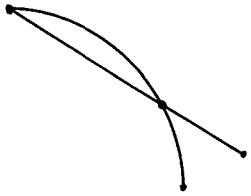
Let E1 and E2 be the 2 edges having 2 intersection points PI1 and PI2, then the routine proceeds as follows:

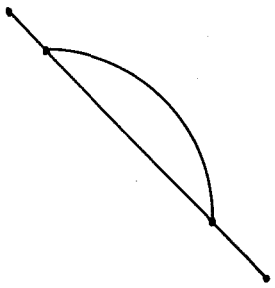
1. A call to the one-intersection routine to handle edges E1 and E2 with one intersection point PI1.

ACTIONS

(1)  2 pairs of coincident vertices. One or two vertices may be deleted.

(2)  1 pair of coincident vertices. One vertex may be deleted. One edge is bisected.

(3)  1 pair of coincident vertices. One vertex may be deleted. One new vertex is created. Two edges are bisected.

(4)  1 edge is trisected.

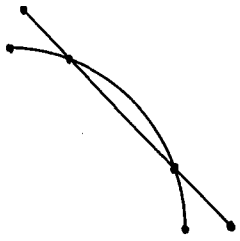
(5)  2 new vertices are created.
2 edges are trisected.

Fig. 4.2 Possible actions taken in the case of two intersection points

2. Check to see if edge E1 has been bisected as a result of step 1. If it has, set E1 as the piece (of the original edge), that now contains the second intersection point PI2.

3. Check to see if edge E2 has been bisected as a result of step 1. If it has, get its pieces and set E2 to be the piece that contains the second intersection point PI2.

4. A call to the one-intersection routine to handle edges E1 and E2 with one intersection point PI2.

5. Exit from the routine.

The advantages of this routine are obvious. Because it can employ another existing routine, it is very simple and neat although the overall possible actions can be so complicated as shown in Fig. 4.2.

4.2.3 Overlapping Edges

This is the most challenging stage in merging. Fig. 4.3 shows possible actions that may be taken in this case.

Let E1 and E2 be the 2 edges concerned with vertices from A1 to B1 and from A2 to B2 respectively. Here, the quintuple $(n_1, n_2, n_3, n_4, n_5)$ would be:

* $n_5 = 3$: to signal the case of overlapping edges.
* n_1 defines the relative position of A2 on edge E1
* n_2 defines the relative position of A1 on edge E2
* n_3 defines the relative position of B2 on edge E1
* n_4 defines the relative position of B1 on edge E2

In addition, another function NW is defined in order to recognise the many possibilities of the case in Fig. 4.3.(2)

$$NW = NW1 + NW2$$

where:

NW1 = 8 if A1 is inside edge E2 and B1 is outside E2

= 9 if B1 is inside edge E2 and A1 is outside E2

= 0 if neither one of the above is true

NW2 = 4 if A2 is inside edge E1 and B2 is outside E1

= 8 if B2 is inside edge E1 and A2 is outside E1

= 0 when neither one of the above is true.

Then if:

NW = 12 would signal that E1 and E2 share the common piece A1A2

= 16 would signal that E1 and E2 share the common piece A1B2

= 13 would signal that E1 and E2 share the common piece B1A2

= 17 would signal that E1 and E2 share the common piece B1B2

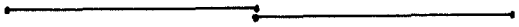
If none of the above cases is true, NW would have a value less than 10.

Then the routine would proceed as follows:

1. Set NW1 = 0 and NW2 = 0
2. If $n_1 \neq 1$, go to step 6
3. If $n_3 \neq 1$, go to step 5
4. At this step $n_1 = n_3 = 1$ or A2B2 is inside A1B1 (Fig. 4.3.(5)).
   Call a subroutine called SPLT31 to trisect edge E1 at points A2 and B2, and also to delete edge E2 and replace it with the middle piece of edge E1.
   Then go to step 6.
5. Here we know A2 is inside edge E1.
   Call a routine called SPLT32 to:
   (a) bisect edge E1 at A2
   (b) take one of the following actions:
   * if $n_3 = 2$: delete either one of vertices B2 and A1 if they are two different vertices. Then delete edge E2 and replace it by the piece A1A2.
   *if $n_3 = 3$: delete either one of vertices B2 and B1 if they are two different vertices. Then delete edge E2 and replace it by the piece B1A2.
   * If $n_3 = 4$: Set NW2 = 4.

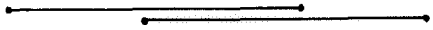
ACTION:

(1)



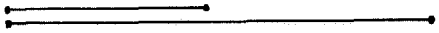
One vertex may be deleted

(2)



Two edges are bisected and one of the two common pieces deleted and replaced by the retained piece

(3)

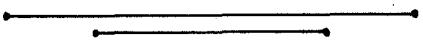


One vertex may be deleted. One edge is bisected and the other edge is deleted and replaced by a piece of the bisected edge

(4)

Two vertices may be deleted. One edge is deleted and replaced by the other edge.

(5)



One edge is trisected. The other edge is deleted and replaced by the middle piece of the trisected edge

Fig. 4.3 Possible actions taken in the case of two overlapping edges

6. If $n_3 \neq 1$, go to step 9.

7. If $n_1 = 1$, go to step 9.

8. Here B2 is inside edge E1.

   Call subroutine SPLT32 to bisect edge E1 at B2 and

   * if $n_1 = 2$: delete either A2 or A1 if necessary.

   Delete edge E2 and replace it by piece A1B2.

   * if $n_1 = 3$: delete either A2 or B1 if necessary.

   Delete edge E2 and replace it by piece B1B2.

   * if $n_1 = 4$: Set NW2 = 8.

9. If $n_2 \neq 1$, go to step 13.

10. If $n_4 \neq 1$, go to step 12.

11. Call subroutine SPLT31 to trisect edge E2 at A1 and B1 and also to delete edge E1 and replace it with the middle piece of edge E2. Then go to step 13.

12. Call subroutine SPLT32 to bisect edge E2 at A1 and:

   * if $n_4 = 2$: delete either B1 or A2 if necessary.

   Delete edge E1 and replace it with piece A1A2.

   * if $n_4 = 3$: delete either B1 or B2 if necessary.

   Delete edge E1 and replace it with piece A1B2.

   * if $n_4 = 4$, set NW1 = 8.

13. If $n_4 \neq 1$, go to step 16.

14. If $n_2 = 1$, go to step 16.

15. Call subroutine SPLT32 to bisect edge E2 at B1 and:

   * if $n_2 = 2$: delete either A1 or A2 if necessary.

   Delete edge E1 and replace it with piece A2B1.

   * if $n_2 = 3$: delete either A1 or B2 if necessary.

   Delete edge E1 and replace it with B2B1.

   * if $n_2 = 4$ set NW1 = 9.

16. Compute NW = NW1 + NW2.

    If NW = 0, go to step 18.

    If NW $\geq$ 12, go to step 17.

    Otherwise, go to step 21.

17. Note that at this step, a situation of Fig. 4.3.(2) exists. Edges E1 and E2 are each bisected and the number of edges are now NSE and NCE. Also edges E1 and E2 must be both straight or both curved. Suppose they are both straight (the step is similar if they are both curved), then edge E1 must have been bisected into (NSE-3) and (NSE-2) while edge E2 must have been bisected into (NSE-1) and NSE. One of the edges (NSE-3) and (NSE-2) must completely overlap one of edges (NSE-1) and NSE. A quick test of comparing the sum of the vertex numbers of these edges would reveal this completely overlapping pair. Delete one of them and replace it with the retained edge. Then go to step 21.

18. Note that at this step, edges E1 and E2 must be either of Fig. 4.3.(1) or 4.3.(4).

    If any one of the numbers n_1, n_2, n_3, n_4 is equal to 4, Fig. 4.3.(1) exists, go to step 20.

19. Fig. 4.3.(4) exists.

    Set FV1 = A2, then set

    FV2 = A1 if $n_1 = 2$

    = B1 if $n_1 = 3$

    and delete whichever vertex is smaller between FV1 and FV2 and replace it with the retained vertex.

    Then, similarly set FV1 = B2 and set

    FV2 = A1 if $n_3 = 2$

    = B1 if $n_3 = 3$

    and delete whichever vertex is smaller between FV1 and FV2 and replace it with the retained vertex. Then delete edge E2 and replace it with edge E1 and go to step 21.

20. If $n_1 = 2$, delete either one of A2 and A1 if necessary and replace it with the retained vertex. Then exit from the routine.

If $n_1 = 3$, delete either one of A2 and B1 if necessary and exit from the routine.

If $n_3 = 2$, delete either B2 and A1, if necessary and exit from the routine.

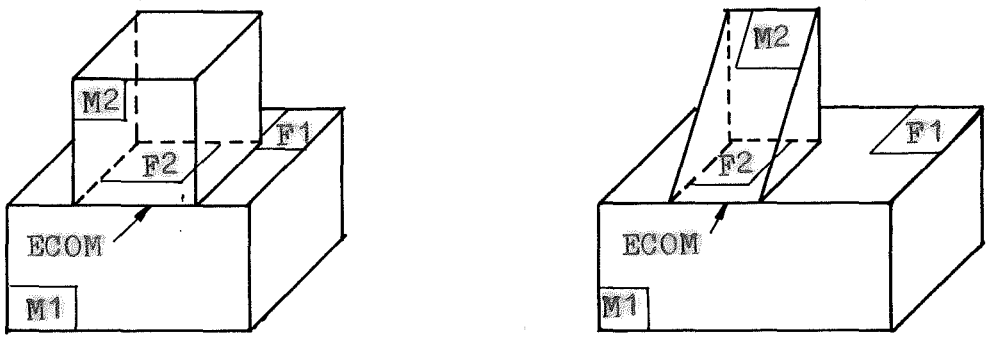
If $n_3 = 3$, delete either B2 and B1, if necessary and exit from the routine.

21. Note that at this stage, one of the edges or pieces of the edges is deleted and replaced by another edge or piece of an edge. Call this replacing edge or piece ECOM, the common edge of face 1 and face 2.

Let face M1 in object 1 meet face F1 at edge ECOM and face M2 in object 2 meet face F2 at edge ECOM, as shown in Fig. 4.4. Then edge ECOM can exist as a "crack" (Fig. 4.4.(1)) when faces M1 and M2 are coplanar and in the same sense, in which case it must be removed. When faces M1 and M2 are not coplanar or are coplanar but in opposite sense, edge ECOM can exist as it is.

This step concerns the removal of a crack when it exists. The routine continues:

Obtain meeting faces M1 and M2 of faces F1 and F2 in object 1 and object 2.



(1) (2)

F1: merging face 1
F2: merging face 2
M1: meeting face 1
M2: meeting face 2.

Fig. 4.4. Combining Meeting Faces

22. If edge ECOM is straight, go to step 24.

23. ECOM is a curve. If both M1 and M2 are cylindrical, mark ECOM as invisible and exit from the routine. If either one of M1 and M2 is cylindrical and the other one conical, leave it as it is and exit from the routine.

    If both M1 and M2 are conical, test to see if there is any change in slope when going from face M1 to face M2 across ECOM and in a direction perpendicular to ECOM. If there is a change in slope, leave ECOM as it is and exit. If there is no change in slope, mark it as invisible and exit.

24. If one merging face is also the meeting face of the other merging face, i.e. M1 = F2 or M2 = F1, a situation which may occur when 2 objects are merged in more than one pair of faces (Fig. 4.5); delete ECOM completely and exit from the routine. If the meeting faces are not the same as merging faces, continue to step 25.

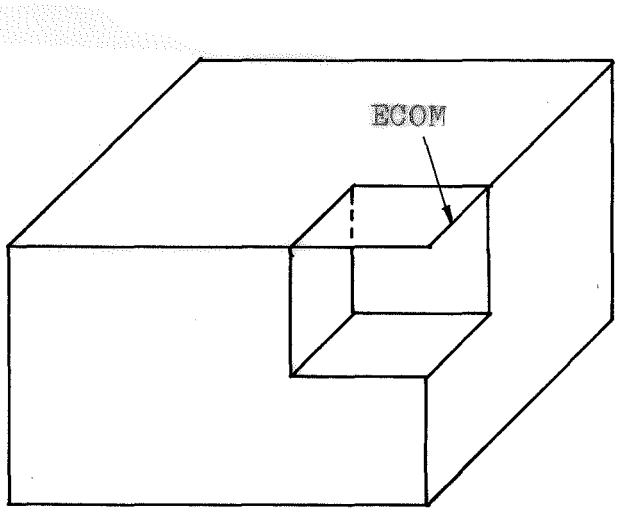


Fig. 4.5 A special case of meeting faces. A small, negative box is merged at a corner of a bigger positive box. This is the 3rd of three successive merges at three pairs of faces.

25. Test to see if there is any change of slope when going from face M1 to M2 across ECOM. If there is no change in slope, i.e. if

$$(A_{M1} - A_{M2})^2 + (B_{M1} - B_{M2})^2 + (C_{M1} - C_{M2})^2 + (D_{M1} - D_{M2})^2 \doteq 0.0$$

continue down to step 26.

If there is a change in slope, exit from the routine.

26. Delete edge ECOM completely.

27. Combining meeting faces

Since meeting faces M1 and M2 are flat and the crack ECOM has been removed, they can be combined to make a new face.

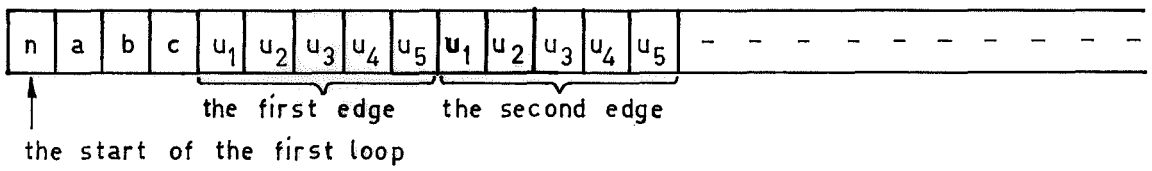
Before trying to combine meeting faces, it is easier to set up a separate simple edge list for each of the faces M1 and M2.

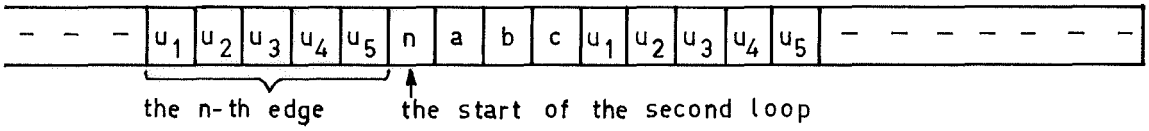
A simple edge list would be described by:

NLOOP: number of loops that exist on the face; the first loop is always the outer perimeter of the face.

ILOOP(K) where K = 1, NLOOP, an array holding the addresses to the edge list ELN of each loop.

ELN: is an array holding the edges belonging to each loop of the face. List ELN has the following form as shown.





In this list, there are NLOOP entries, each entry holds the edges of a loop. The start of each entry is at an address held in ILOOP. For example, a particular loop K on the face has address NS = ILOOP(K). Then:

ELN(NS) = n: the number of edges in the loop

ELN(NS + 1) = a: a flag. If

a = 0: the current loop is a perimeter

= 1: the current loop is a hole.

ELN(NS + 2) = b: intersection flag.

If b = 0: loop has no intersection with any other loop

= 1: loop has intersections.

ELN(NS + 3) = c: if c = 0: loop has not been used

= 1: loop has been used.

and then followed by a group of n quintuples (u_1, u_2, u_3, u_4, u_5) each of which describes an edge. The appearance of edges is still in the strict order as an ordinary EL list.

For i = from 0 to (n - 1)

ELN(NS + 4 + 5 i) = u_1: edge type (see Appendix B)

ELN(NS + 5 + 5 i) = u_2: edge direction (see Appendix B)

ELN(NS + 6 + 5 i) = u_3: the edge number. If u_3 has been used or deleted, u_3 is made negative

ELN(NS + 7 + 5 i) = u_4: the starting vertex of the edge u_3

ELN(NS + 8 + 5 i) = u_5: special vertex flag.

If u_5 = 0: the starting vertex u_4 is not special

= 1: the starting vertex u_4 is special.

The edges contained in list ELN are not split. If any edge on the face is split, it is immediately replaced by its pieces in the correct order in ELN list.

Construct simple edge lists EL1 and EL2 for faces M1 and M2 respectively and continue the routine.

28. Mark face M1 as deleted.

29. If face M2 is the same as M1, go to step 36.

30. Mark face M2 as deleted.

31. If ECOM lies on a hole in face M1, go to step 34.

32. If ECOM lies on a hole in face M2, go to step 35.

33. Here ECOM lies in the outer perimeter of each of the faces M1 and M2 (Fig. 4.6). Make an edge list of a new face to be entered into the EL list. The outer perimeter of the new face is made by adding edges on the perimeter of face M1, starting at any edge on face M1 which is not ECOM, until ECOM is met, (signalled by its starting vertex being a special vertex).

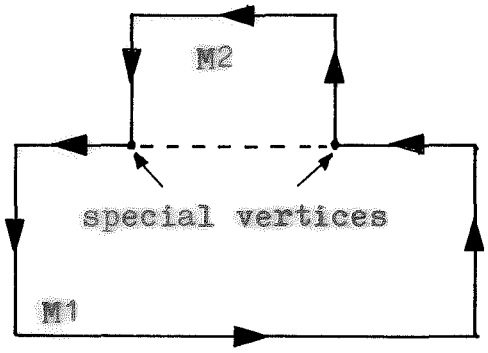


Fig. 4.6. ECOM (shown as dashed line) lies on the outer perimeters of both faces.

Switch on to the EL2 list of face M2 and look for an edge having the same starting vertex. Continue to add this edge and those that follow on to the new perimeter until another special vertex is met. Switch back again to EL1 list and look for an edge starting at this vertex. Add this edge to the new perimeter. Continue to add edges until the new perimeter is closed i.e. the very first starting vertex is met. Copy all holes in EL1 and EL2 lists as holes of the new face.

Then go to step 38.

34. ECOM lies on a hole in face M1 and on the perimeter of face M2. Call a subroutine called COMF2 to handle this case (see in step 35). Then go to step 38.

35. ECOM lies on a hole in face M2 and on the perimeter of face M1. Call COMF2 to handle this case. Then go to step 38.

Subroutine COMF2

This subroutine handles the case where ECOM lies on a hole in one face and on the perimeter of the other face (Fig. 4.7). Copy the perimeter on which ECOM does NOT lie as the new perimeter of the new face. Then copy the other perimeter until a special vertex

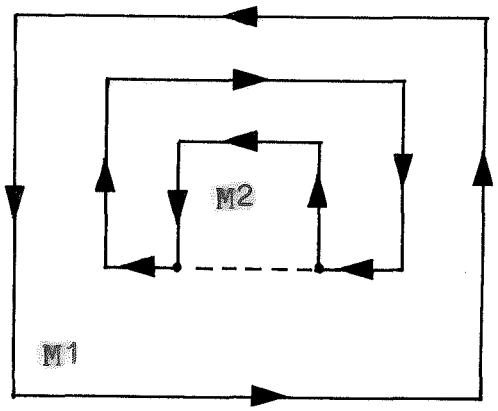


Fig. 4.7

is met; switch to the hole containing ECOM and add the edge starting at this vertex; continue adding edges until another special vertex is met; switch back to the perimeter and add the edge starting at this vertex; continue to add edges until the loop is closed. The loop just formed is a hole of the new face. Copy the remaining holes of each face as holes of the new face. Then return from the subroutine.

36. Only one face is involved.

If ECOM lies on a hole in the face, go to step 37.

ECOM lies on the perimeter of the face (Fig. 4.8)

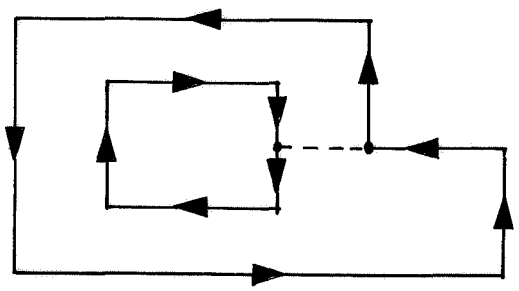


Fig. 4.8

In this case the perimeter becomes two loops. Start from any edge which is not ECOM in the list, add edges until a special vertex is met. Look for another edge in the face that starts at this vertex and add this edge to the loop. Continue adding edges until the loop closes. This is one loop. Note that any edge which has been added is marked "used" so that it will not be used again. Similarly start from any other un-used, non-deleted edge in the face, add edges to the second loop until a special vertex is met. Look for another edge that starts at this vertex and add this edge to the loop. Continue adding edges in this way until the loop closes.

Use a subroutine called INSPEC (described in Braid's [1]) to determine which loop surrounds the other. Copy the outer loop as the new perimeter of the new face. Copy the inner loop as a hole. Copy all other holes as holes of the new face.

Then go to step 38.

37. ECOM lies in a hole which now becomes 2 holes which must be disjoint (Fig. 4.9)

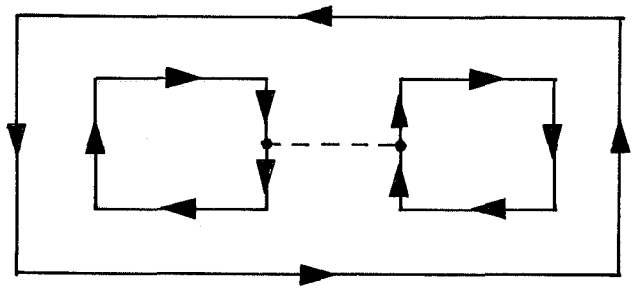


Fig. 4.9

The splitting of this hole into 2 new holes is similar to the splitting of a perimeter into 2 new loops as described in step 36 above. Therefore, copy the perimeter of the face as the new perimeter of the new face. Copy 2 new holes as holes of the new face. Copy all other holes as holes of the new face.

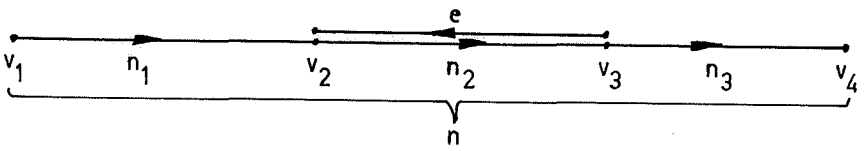
38. Enter the new face into the EL list and update the date structure.

39. Exit from the routine.

It should be noted that only flat faces can be combined because the implicit form of the flat face equations can be extended indefinitely. Curved faces cannot be combined because parameters of their face equations have certain limits which cannot be extended.

A Note about Edge Deletion

Consider straight edges. When an edge is split, its pieces are stored in a tree-type structure down the EVS list. The direction of its pieces is the same as the direction of the parent edge and only the direction code of the parent edge is held in the EL list. Suppose an edge n is trisected into n_1, n_2, n_3 whose directions are from vertex v_1 to v_2, v_2 to v_3, v_3 to v_4 respectively. Suppose later on, edge n_2 is deleted and is replaced by another edge e whose direction is now from v_3 to v_2. Then when edge n is scanned, its pieces are fetched, which are now n_1, e, n_3 whose directions are from v_1 to v_2, v_3 to v_2, v_3 to v_4!! So the difficulty arises when an edge is deleted and replaced by another edge of opposite direction.



Original direction of edge n is from v_1 to v_4.

Two solutions are in sight:

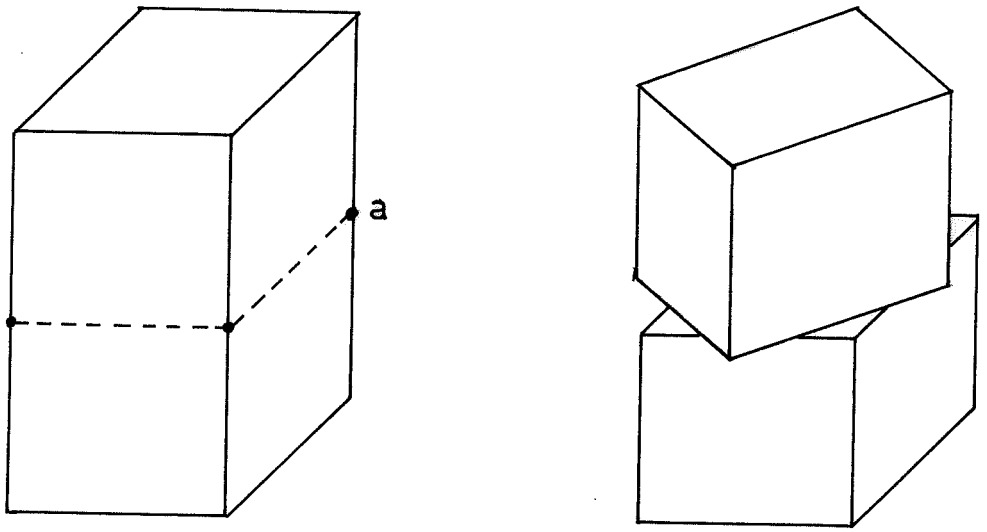
1. either change the direction of e in the EVS list so that it has the same direction as n_2, and then change the direction code of edge e in EL list;
2. or change the direction of n, n_1, n_2, n_3 in EVS list so that n_2 has the same direction as e and then change the direction code of edge n in EL list.

So it appears that the first solution is preferred because only the

direction of e is changed. However, e can be a piece of another split edge and if its direction is changed, the directions of other pieces and its parent edge must also be changed in the same way as for the second solution. In this case, a further step in fetching the parent edge of e and its pieces would be taken. Therefore, the second solution is adopted here.

4.3 Stage Three

At the end of the second stage, all intersections of edges on the merging plane are found, all intersecting edges are split and meeting faces are combined. The third stage joins up edges or pieces of edges on the merging plane to make a new face or new faces. There might not be any new faces at all on the merging plane (Fig. 4.10)



(1) No new faces are formed (2) Eight new faces are formed

Fig. 4.10

The procedure for the third stage is:

1. Make simple edge lists EL1 and EL2 for faces F1 and F2 respectively.
2. Calculate the signed area of the outer perimeter of face F1, say, as projected onto a plane which is not orthogonal to the plane of the face. This plane is chosen as a plane whose normal is in the direction of the largest direction cosine of the unit normal of the

face.

e.g. the equation of face F1 is

$$Ax + By + Cz + D = 0.$$

Suppose $|C|$ is greatest of all $|A|$, $|B|$, $|C|$ then x-y plane is chosen as the projection plane. Then the signed area of face F1, projected on xy plane is given by:

$$\text{area} = \sum_{i=1}^{(n-1)} \tfrac{1}{2} (x_i y_{i+1} - x_{i+1} y_i)$$

n: number of vertices of the loop.

The sign of the area is important because it reveals the type of the loop and the type of the face.A positive face would have positive area for its outer perimeter and negative areas for all its holes. Note that this formula applies when curve edges are involved as well, although the area calculated will not be the exact projected area, the sign is still the same.

3. Mark faces F1 and F2 as being deleted.

4. If the outer perimeter of face F1 has no intersection i.e. EL1(3) = 0, go to step 7.

5. If the outer perimeter of face F2 has no intersection i.e. EL2(3) = 0, go to step 9.

6. Outer perimeter of F1 intersects the outer perimeter of face F2. Call a subroutine called COM323 to make new faces. Choose any unused edge on the perimeter of face F1 to start a new face. Add edges to it from the current perimeter until a special vertex is met. Record the starting vertex of the first edge chosen to start the face and every edge which has been added is marked "used" so that it will not be used again. As edges are added, the area traced out by them as projected on the projection plane is kept. When a special vertex is met, swap to the other perimeter, i.e.

that of F2 and look for an edge starting at this vertex and add it to the new face. Continue adding edges in this way, swapping to the other perimeter every time a special vertex is met. Adding is stopped when the first starting vertex is met, i.e. the current loop is closed. The loop just formed is the outer perimeter of the new face. The new face has the face type (positive or negative) of face F1 if their projected areas have the same sign. Otherwise, the new face has the face type of face F2. Look for all un-used, non-intersecting holes of the merging face (F1 or F2) of the same type and add them as holes of the new face if INSPEC shows that they are within the new face. Every time a hole is added, it is marked "used" so that it will not be used again. The new face is completed and is added to the data structure.

If there are any more un-used edges in the perimeter of face F1, go back to the beginning of the subroutine to make another new face. If all edges in the perimeter of F1 have been used, mark perimeters of F1 and F2 as "used".

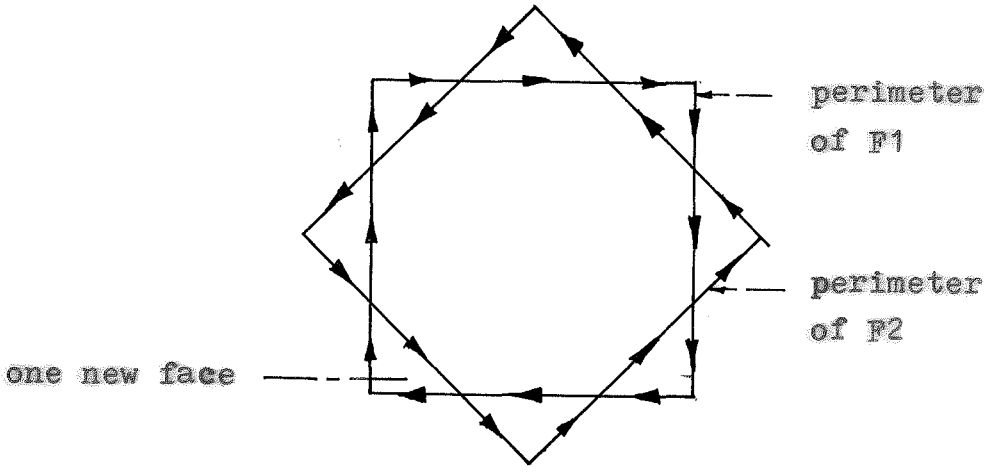


Fig. 4.11 Intersecting perimeters. Eight new faces are formed.

Then go to step 12.

7. If the outer perimeter of face F2 has no intersection, i.e. EL2(3) = 0, go to step 10.

8. Perimeter of face F2 intersects a hole(s) of face F1. Call a

subroutine called COM 367 to deal with the case when one perimeter intersects a hole of the other perimeter (Fig. 4.12).

Subroutine COM367

(a) Choose the outer perimeter as the perimeter of the new face having the same face type

(b) Examine all un-used, non-intersecting holes of the original outer face to see which ones lie outside the perimeter of the inner face, again using INSPEC. Add these holes as holes of the new face.

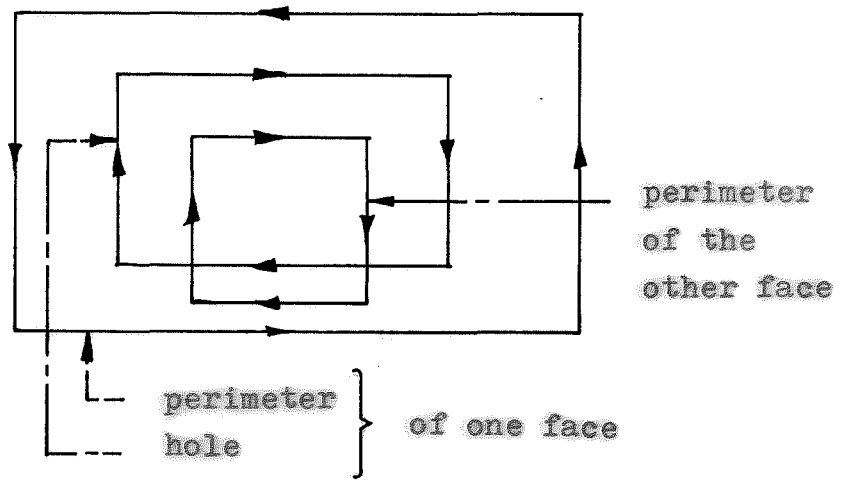


Fig. 4.12 One perimeter intersects.

(c) Examine each un-used, intersecting hole of the outer perimeter. Look for a non-special vertex on this hole which lies outside the inner perimeter. From this vertex, start a new hole for the new face. Add edges to it, swapping to the other face every time a special vertex is met and look for an un-used edge starting at this vertex and add it to the hole. Keep adding edges until the new hole closes.

(d) If there are any more un-used, non-special edges on the intersecting holes of the original outer perimeter which lie outside the inner perimeter, go back to step C above to start another

new hole

Otherwise, the hole just formed is the last hole of the new face. Then go to step 11.

9. Perimeter of face F1 intersects hole(s) of face F2. Call subroutine COM367 to deal with this case. Then go to step 11.

10. Neither perimeter intersects. In this case one perimeter must surround the other. Again use INSPEC to determine which perimeter surrounds the other. This test is quite simple. Test each vertex on the perimeter of face F2, say, to see if it is inside or outside the perimeter of face F1 (since INSPEC can only test the position of one point with respect to a closed loop). If there is a vertex at any stage lying inside, face F2 must lie inside face F1. Otherwise face F1 must lie inside face F2.

    (a) Copy the outer perimeter as the perimeter of the new face.

    (b) Copy all un-used, non-intersecting holes of the original outer perimeter which lie outside the inner perimeter, as holes of the new face.

    (c) Copy the inner perimeter as the final hole of the new face.

    Then continue down to step 11.

11. The new face is completed. Add it to the data structure.

12. If all holes in each face have been used, the third stage is completed.

13. If there are no more un-used, intersecting holes on either face, go to step 15.

14. Un-used, intersecting holes on either face.

    This case is very similar to step 6 because the intersecting loop now is a hole, not a perimeter. Call subroutine COM323 repeatedly if necessary to make new faces until every un-used, intersecting holes on each face have been dealt with.

15. If there are no more un-used, non-intersecting holes on either face,

the third stage is completed.

16. Un-used, non-intersecting holes on either face

For each hole in face F1, test with each of the holes of face F2:

(a) If that hole of face F1 surrounds the other hole of face F2, the hole of face F1 is made the perimeter of a new face having the face type of face F2. The hole in face F2 is added as a hole to the new face. Add as holes to the new face any more holes on face F2, which lie inside the new perimeter.
Go to step d.

(b) If that hole of face F2 surrounds the other hole of face F1, the hole of face F2 is made the perimeter of a new face having the type of face F1. The hole in face F1 is added as a hole to the new face. Add as holes to the new face any more holes on face F2, which lie inside the new perimeter.
Go to step d.

(c) If neither the case of (a) nor (b) is true for that hole of face F1 after being tested with all holes of face F2, that hole of face F1 is the perimeter of a new face which has the type of face F2 and which has no holes.

(d) If there are any more holes on face F1 to be considered, go back to step (a) and repeat until all holes in face F1 have been dealt with.

17. If there are no more un-used, non-intersecting holes on face F2, the third stage is completed.

18. Un-used, non-intersecting holes on face F2

Enter each of these holes as a perimeter of a new face. Each new face has the type of face F1 and has no holes.
The third stage is completed.

It should be pointed out that in the third stage, an "empty face" might be made, which results from adding edges to it alternatively from one face and the other. An example is shown in Fig. 4.13, which is the merging plane of Fig. 4.4.(2)

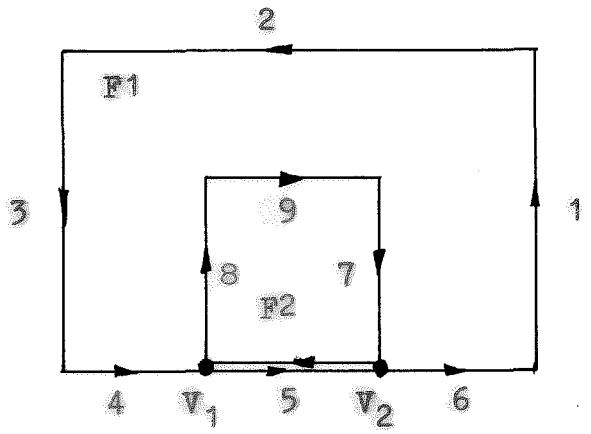


Fig. 4.13 An "empty" face.

In this example edge 5 is not a crack and hence not removed but is common to both faces.

Face F1 would have edges, in sequence: 1, 2, 3, 4, 5, 6 with vertices V_1 and V_2 special.

Face F2 would have edges: 7, 5, 8, 9 again with vertices V_1 and V_2 special.

Now if edge 1 starts a new face, the sequence of the edges being added is:

* edges 1, 2, 3, 4 from face F1 when V_1 is met.
* edges 8, 9, 7 from face F2 when V_2 is met.
* edge 6 from face F1 when the first starting vertex is met and the loop closes.

The face just formed is perfect, but now in face F1, edge 5 remains un-used and hence an attempt to make another new face is made. The sequence is:

* edge 5 from face F1, starting vertex at V_1 and adding stops

because V_2 is special. Swap to face F2.

* edge 5 is added from F2 since edge 5 in face F2 starts from V_2 and ends at V_1. The face closes perfectly because vertex V_1 is met!

Fortunately, since a running total of the area of the face is always kept, any "empty" face would be signalled by its area being equal to zero. And in this case, the new face just formed is immediately ignored and the procedure continues.

4.4 Stage Four

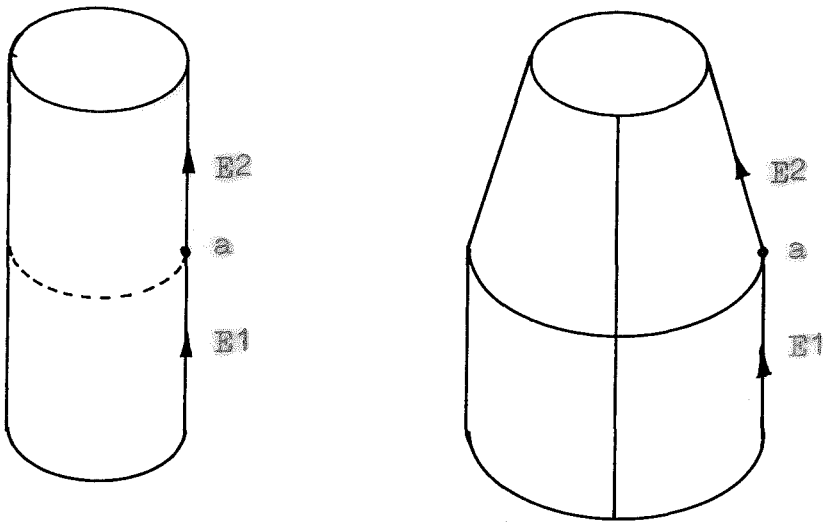
The fourth stage is a "garbage-collection" stage. The complete data structure is tidied up. Vertices, edges, faces may be re-numbered. Features which are marked "deleted" are thrown out. Split edges are replaced by their pieces, etc. The final form of the data structure would be like that described in appendix B in which no features are marked "deleted", or "split", or "replaced".

However, before commencing the fourth stage, it might be best to clear one special case which might result from the merging procedure. Consider 2 identical boxes merged in the manner shown in Fig. 4.10.(1). The dashed lines are those of the original boxes and are removed as the result of merging. However, vertices such as "a" still remain in the data structure and are not necessarily desirable. The reason is because "a" just joins 2 colinear, straight edges together. These 2 straight edges can be joined into one single straight edge and vertex "a" is removed, making the data structure better. Note that 2 curved edges cannot be joined into one in this way because the limits of the parameter might be exceeded. The procedure to remove a vertex joining 2 straight, co-linear edges and join the 2 edges together is as follows:

1. Get the next vertex I of the object. Check EVS list to see how many edges it lies on. If it is on more than 2 edges, go to step 6.

2. Vertex I is on 2 edges in EVS list. Call E1 the edge which ends at I and E2 the edge which starts at I.

3. Check through the EVC list to see if any curved edge contains I. This step is necessary because if it is true, the vertex cannot be removed as shown in Fig. 4.14.



(1) (2)

Fig. 4.14

If this is true, go to step 6.

4. Mark edge E2 as deleted.

5. In edge E1 substitute vertex I by the vertex at which edge E2 ends. Therefore edge E1 is now extended to cover edge E2. Delete vertex I.

6. If there are any more vertices to be considered, go to step 1. Otherwise the procedure is complete.

Then stage four can commence:

1. Check all flat faces to see if any one is marked as deleted. Every time one is found, a call to subroutine DELF is made to remove that face from the data structure.

Subroutine DELF works as follows:

(a) The face equation of the deleted face is removed from the array AFF by moving every equation of faces that follow up one entry to over-write it. AFF now contains only (NFF-1) equations. If the face is curved (e.g. in intersection algorithm), the array AFC is used instead of AFF.

(b) The data block of its edges held in EL list is also removed from the array EL by moving all entries that follow it up the list to over-write it.

EL now holds (NF-1) entries and is (NEL-length of the deleted block) long.

(c) Face status array is also updated correspondingly.

(d) Replace NEL by its new length.

Reduce NF, NFF by 1 each. If the deleted face is curved, NFF is unchanged.

(e) Exit from the routine.

2. Check EVC and EVS to see if there is a split, or a deleted or a replaced edge. If there is not any such edge, the fourth stage is complete. Otherwise, set ELD = 0.

3. Get the next face of the object.

Get the address to the edge data block of that face in EL list. This address is updated by adding ELD to it.

4. Get the next edge of the current face from EL list. Note if it signals the start of a hole, a straight or curve edge and its direction.

5. If the current edge is not replaced or deleted, go to step 9.

6. If the current edge is deleted, go to 8.

7. The current edge is replaced. Get the replacing edge and make it the current edge, then go to step 5.

8. Move the data groups of the edges that follow the deleted edge

up the list to over-write and thus remove the deleted edge from the list. If the deleted edge is the start of a hole, the edge that immediately follows it is made the start of the hole.

Reduce the number of edges of the current face by 1. Reduce ELD, the offset of the length of EL list, by 4 (note that each edge data is held in 4 spaces in EL). Then go to step 15.

9. Store the current edge number back in its correct place in EL list.
10. If the current edge is not split, go to step 15.
11. Get all the pieces of the current edge, which have not been deleted.
12. Push down EL list at the address of the current edge to make enough room to store all its pieces in the list. Increase ELD by the number of spaces being pushed down.
13. Enter each piece of the current edge into the space just made in their correct sequence. If the current edge starts a hole, the first piece that is entered starts the hole.
14. Increase the number of edges of the current face by (the number of pieces of the current edge - 1).
15. If there are any more edges of the current face to be considered, go to step 4.
16. Update and store the number of edges of the current face at its right place in EL list.
17. If there are any more faces to be considered, go to step 3.
18. EL list has been dealt with. Update its length NEL by adding ELD to it.
19. For each undeleted, unreplaced, non-split edge in the EVS list, check to see if it appears in the face-edge (EL) list. If it does not, mark it as deleted in the EVS list.

    Repeat this step for all edges in the EVC list if there are curved edges.

20. For each vertex, check to see if it is on an undeleted, un-replaced, non-split edge in the EVS or EVC lists. If it does not, delete it and update the EVS and EVC lists.
21. Go back to the beginning of EVS and EVC lists.
22. Get the next edge from the EVS list.
23. If the current edge is non-split, or not deleted or not replaced by any other edge, go to step 26.
24. Move all edges that follow the current edge up one entry in the EVS list to over-write the current edge. Reduce NSE by 1.
25. Run through EL list and look for all straight edges which have numbers higher than the number of the current edge. Reduce each of these edges' numbers by 1.
26. If there are more edges from the EVS list to be considered, go to step 22.
27. If there are no curved edges, go to step 31.
28. Repeat steps 22 through to 26 for the EVC list, the list of curved edges.
29. Scan the L-number list to look for a group of L-numbers which no longer correspond to any curved edges:
    (a) When such a group is found, it is deleted from the L-number by moving all the entries that follow it up one entry to over-write it. Also scan the EVC list to look for all L-number groups which are numbered higher than the number of the group just deleted. Reduce each of these groups' numbers by 1.
    (b) If no such group can be found, go to step 31.
30. Repeat step 29 for the parameter limit list PL.
31. The fourth stage is complete.

At the end of this stage, the date structure of the object is at its minimal size, i.e. as described in appendix B.

CHAPTER 5

INTERSECTION

Intersection performs addition and subtraction between two intersecting objects which do not necessarily have to have any flat, coplanar faces. The basic intersection algorithm finds the object resulting from the intersection of a negative object and a positive object. The resulting object has all the material inside the positive object and outside the negative object. However, as a corollary, intersection is found capable of finding the union of two positive objects. To do this, the two positive objects are negated first, then intersected and the resultant object is negated again.

Intersection algorithm is initiated by running the batch file TGINTN and then enter:

record number 2 (of object 2), record number 1 (of object 1)

where object 2 is the negative object and object 1 is the positive object. It is the object 2 which is added to or subtracted from the object 1.

The general procedure is to examine all possible pairs consisting of an edge from one object and a face from the other object. All intersections between the edge-face pairs are found and recorded. Every time an intersecting edge is broken up, one piece is deleted and the other piece is retained. New edges, being the intersection lines between intersecting faces, are created. Then on each intersecting face, the original edges and undeleted pieces of the original edges are joined together with new edges to make a new face or new faces. Finally, a test has to be made to decide which non-intersecting faces of the second object are to be retained in the final object and which are to be excluded.

The characteristics of intersection are that new vertices, new edges and new faces can be created and an original face can be completely deleted without being replaced by any new faces. It should be pointed out that curved faces are not allowed to intersect because their intersection may result in a curve which cannot be represented in this data structure.

In Braid's work, there was a restriction that the second object, the negative object, had to be a primitive box or a primitive cylinder. The reason for this restriction was to be able to enclose the second object in a sphere of a known diameter. This sphere was then tested with the first object to reveal immediately the faces and edges which do not intersect the sphere and thus the second object. This knowledge could eliminate unnecessary testing of non-intersecting edge-face pairs and hence speed up the algorithm. In the present work, this restriction is omitted. When the second object is found to be a primitive, the sphere test is carried out. When it is not a primitive, the sphere test is by-passed. In doing this, the algorithm is slow, but all general objects can be intersected.

Like merging, intersection has four stages.

5.1 The First Stage

Like merging, the first stage is to coalesce the data structures of the two objects. If the second object is a primitive, it is surrounded by a sphere which will be tested for possible intersections with every face of the other object. This will eliminate all faces and edges which do not involve in the rest of the algorithm.

To simplify the algorithm, the coalescent object is translated so that the centre of the sphere is at the origin of the co-ordinate system. If the second object is a general object, four arbitrary vertices of it are chosen to define a sphere. The coalesced object is then similarly translated as if the sphere enclosed the second object. In either case,

the origin of the co-ordinate system is a point inside the second object, a knowledge which becomes very handy in the third stage.

The procedure is:

1. Read the data structures of object 1 and of object 2. Save the 8-number header (NP1, NF1, NFF1, NSE1, NCE1, NEL1, NL1, NPL1) of object 1 so that its features (e.g. faces and edges) can be deduced from the coalesced structure.
2. Coalesce the two data structures in exactly the same way as in the first stage of merging.
3. If the second object is a primitive, calculate the centre and the radius of the sphere surrounding it (Section 5.1.1.)
   If the second object is a general object, a sphere enclosing its four arbitrary vertices is calculated.
4. Form a translational matrix to bring the centre of the sphere to the origin. Apply this transformation to the coalesced object.
   If the second object is a general object, go to step 7.
5. Let F1 be a surface on which a face of object 1 lies.
   Test F1 for possible intersections with the enclosing sphere.
   (a) F1 is a flat face:
   The normalized equation of F1 is:
   $Ax + By + Cz + D = 0$
   where $|D|$ is simply the distance from F1 to the origin.
   If $|D|$ > the radius of the sphere, mark F1 as non-intersecting by setting FS(2,F1) = -1.
   Also mark every edge E lying on face F1 as non-intersecting by setting EVS(4,E) = -1 or EVC(7,E) = -1 depending on whether edge E is straight or curved.
   If $|D| \leq$ the radius of the sphere, intersection is possible.
   Test further to see if F1 is coplanar with any other face F2 of

object 2. If one such face F2 is found, mark them as co-planar.

(b) F1 is a curved face:

Braid suggested that F1 should be replaced by 3 bounding flat faces in a similar manner to a curved edge being replaced by a triangle (see section on tangent crossing point). Then each of these flat faces is tested with the sphere and when they are all outside the sphere, another test to see if the origin (i.e. centre of the sphere) is outside the space bounded by the three flat faces is necessary before the face can be marked as non-intersecting. This method involves quite a lot of work in calculating the tangent crossing points, setting up the face equations of the flat faces and especially in determining whether the origin is outside the space surrounded by the three flat faces. And yet is is still inefficient because in many cases the sphere cuts one of the three flat faces and still might not cut the curved face.

Another method is adopted here. The point P lying on the curved face F1 and approximately nearest to the origin is calculated. (See Section 5.1.2). Then if the distance from P to the origin is greater than the radius of the sphere, F1 and its edges are marked as non-intersecting.

6. Repeat step 5 for every face of object 1. The first stage is complete.

7. Test every flat face of object 1 with every other flat face of object 2. If a pair of coplanar faces is found, mark them as coplanar.

The first stage is complete.

5.1.1 Finding the Sphere enclosing a primitive

A sphere in space can be defined in so many different ways depending on the shape of the object to be enclosed. Here a unique definition, if possible, is required for a sphere enclosing any one primitive object. Such a definition is possible when the data structure of each of the primitives is closely examined. Each primitive has vertices 2, 3, 4, 5 being non-coplanar and therefore they can define the enclosing sphere.

Let P(x, y, z) be the centre of the sphere, then the centre of the sphere is found by:

$$\overrightarrow{P_2P} \cdot \overrightarrow{P_2P_3} = \tfrac{1}{2} \left|\overrightarrow{P_2P_3}\right|^2 \quad (1)$$

$$\overrightarrow{P_2P} \cdot \overrightarrow{P_2P_4} = \tfrac{1}{2} \left|\overrightarrow{P_2P_4}\right|^2 \quad (2)$$

$$\overrightarrow{P_2P} \cdot \overrightarrow{P_2P_5} = \tfrac{1}{2} \left|\overrightarrow{P_2P_5}\right|^2 \quad (3)$$

and the radius of the sphere would be $\left|\overrightarrow{P_2P}\right|$ where P_2, P_3, P_4, P_5 are vertices 2, 3, 4, 5 respectively.

5.1.2 Finding the point which lies on a curved face and which is approximately nearest to the origin

Since a point is expressed in terms of a pair of 2 parameters (s, t) on a curved face where $0 \leq s \leq 1$ and $0 \leq t \leq 1$, the procedure is quite simple by keeping one parameter constant and varying the other until the distance $\left|\overrightarrow{OP}\right|$ between the origin 0 and the point P is minimum and the same process is applied to the other parameter.

The exact procedure goes as follows:

1. Set t = s = 0.00 and calculate point P (s, t) on the curved face.
2. Increase s by a small fraction, in practice, it is 0.01 i.e.

   set s = s + 0.01.

   If s > 1.0 go to step 5.
3. Calculate point P_1 (s, t) on the curved face.

4. If $|\overrightarrow{OP}_1| > |\overrightarrow{OP}|$ go to step 5

   If $|\overrightarrow{OP}_1| \leq |\overrightarrow{OP}|$, then let $\overrightarrow{OP} = \overrightarrow{OP}_1$

   and go to step 2.

5. Again put t = t + 0.01.

   If t > 1.00 go to step 8.

6. Calculate P_1 (s,t)

7. If $|\overrightarrow{OP}_1| > |\overrightarrow{OP}|$, go to step 8

   If $|\overrightarrow{OP}_1| \leq |\overrightarrow{OP}|$, set $\overrightarrow{OP} = \overrightarrow{OP}_1$ and go to step 5.

8. P is the nearest point, approximately, and the procedure is completed.

   It is worth noting that along any constant t or constant s line, there is only one minimum $|\overrightarrow{OP}|$.

5.2 The Second Stage

The second stage finds all intersections between two objects. In merging, each edge of one object is tested against all coplanar edges of the other object. In intersection, each face of one object is tested against all edges of the other object provided that the face or the edge has not been marked as non-intersecting.

The procedure for selecting the face-edge pairs is:

1. Get the next face of the second object. If there are no more faces left, go to step 6.

   If the face is flat, go to step 2.

   If the face is cylindrical, go to step 4.

   If the face is conical, go to step 5.

2. Call a routine to find the intersection of the flat face and every unmarked, straight edge of the other object. If an edge has already been split, get all its pieces and test each piece separately against the face (Section 5.2.2.). Split the edge if necessary and record the intersection in the IVEF list (Section 5.2.1.).

3. If the first object has no curved edges, go to step 1.
   Repeat step 2 with every unmarked, curved edge of the first object. Go to step 1.
4. Call a routine to find the intersection of the cylindrical face and every unmarked, straight edge of the other object. Record all intersections as in step 2. Go to step 1.
5. Call a routine to find the intersection of the conical face and every unmarked, straight edge of the other object. Record all intersections as in step 2. Go to step 1.
6. Repeat from step 1 to step 5 for every unmarked face of object 1 and every edge of object 2. The second stage is complete.

It should be noted that intersections between curved faces and curved edges are not considered because they are not allowed in this data structure.

5.2.1 The IVEF list

In merging, the information about edge intersections is fully stored by the splitting of the edges and all pieces are recorded in the same list in a tree form. In intersection, the intersection occurs both on an edge and on a face. The information about the edge is still adequately stored by the splitting of that edge as described in merging. But the information about the face cannot be stored within the data structure. A separate list, called the IVEF list, has to be made to store this information.

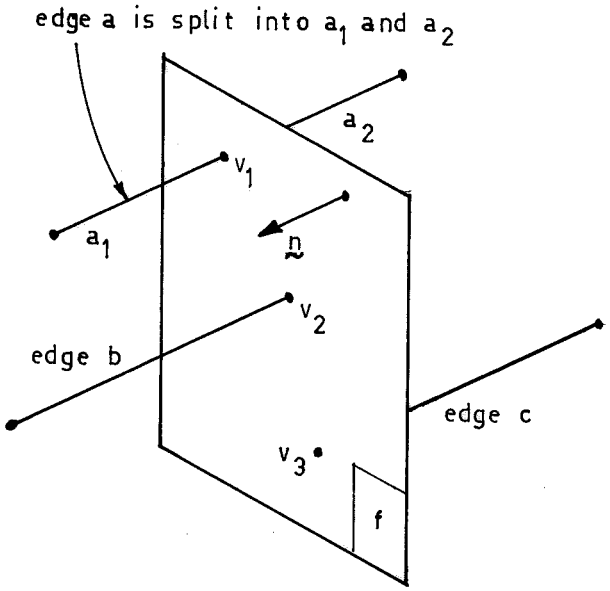
Note that in splitting an edge and recording its pieces in a tree form, its pieces when retrieved are in the correct order, i.e. they are in the same direction as the parent edge. In the IVEF list each intersection is entered as it is found and there is no question of order at this stage. It is not until the third stage when this problem is sorted out. The IVEF list must be kept separately and passed on to the third stage since it is not within the data structure

of the object.

The IVEF list is a two dimensional array. Each entry has 4 elements as shown in Fig. 5.1.

n_1: Contains the vertex number where the intersection occurs

n_2: contains the edge number of the edge piece or of the edge which lies "inside" the object and which will be retained. An edge is considered as lying inside the object if it is on the opposite side to the normal vector of the face.



n_1	n_2	n_3	n_4
v_1	a_2	f	a
v_2	0	f	b
v_3	c	f	c
—	—	—	—
—	—	—	—
—	—	—	—

Fig. 5.1. The IVEF list

In this example, a_2 of the original edge a and the whole edge c are retained while the whole edge b is discarded ($n_2 = 0$) because it lies "outside" the object.

n_3: contains the face number of the face

$\varepsilon(0,100)$ for flat faces

$\varepsilon(100,200)$ for cylindrical faces

$\varepsilon(200,300)$ for conical faces.

n_4: contains the original edge number of the edge

$\varepsilon(0,200)$ for straight edges

$\varepsilon(200,300)$ for cylindrical edges

$\varepsilon(300,400)$ for conical edges

5.2.2 The Intersection of a Face and an Edge

Formulae for finding intersections between flat face - straight edge, flat face - curved edge and curved face - straight edge are given in appendix E. A general procedure for finding face-edge intersections is:

1. Determine which of the (Ox, Oy), (Oy, Oz) and (Oz, Ox) planes is the projection plane i.e. the one not perpendicular to the face F. This plane will be used for subroutines "INPEC" or "CINSPE" later.
2. Get the edge, note if it has been split. Put the edge or its pieces in an IL list. Set the pointer of the list ILP = 1.
3. Set E as the edge in the IL list, indicated by ILP. If there are no more edges, exit from the routine.
4. Find the intersection between the face F and the edge E using one of the formulae given in appendix E. If there is no intersection, go to step 6.

   If there are intersections, check that the intersection points lie on the finite edge E and also within the perimeter, but not in a hole, of the face F (using INSPEC or CINSPE). Record the intersections in a group of 7 integers $(q_1, q_2, q_3, q_4, q_5, q_6, q_7)$ (Section 5.2.2.1.)

   If the edge E lies on the face F, exit from the routine.
5. Split the edge and record the intersection according to the q-numbers (5.2.2.2.).
6. Set ILP = ILP + 1 and go to step 3.

Note that when an edge E is coplanar with the face F, it is still necessary to find all intersections between the edge E and all other

edges lying on the face F as Braid did. However, in the above algorithm, whenever an edge E is found to lie on a face F, it is skipped. This does not mean that the intersections between E and all edges of F are missed out. Since every edge is the intersection of two non-coplanar faces, every edge e of face F must also lie on another face f. In other words, face f is the meeting face of face F at e. Therefore, the intersection, if any, between edges E and e will certainly be revealed when the face f-edge E pair is tested. This observation is very useful because it reduces a considerable amount of work in stage 2.

5.2.2.1 Describing an Intersection

In merging, intersections are described by a quintuple $(n_1, n_2, n_3, n_4, n_5)$. In intersection, the intersections are described by 7 integers $(q_1, q_2, q_3, q_4, q_5, q_6, q_7)$ where:

q_1: gives the position of the first intersection point on the edge.

If $q_1 = 1$: point is on the edge

$q_1 = 2$: point is the first vertex of the edge

$q_1 = 3$: point is the second vertex of the edge

$q_1 = 4$: point is outside the finite edge

q_2: gives the position of the first intersection point on the face.

If $q_2 = 0$: point is inside the face or on the perimeter of the face but not on a vertex of the face

$= n > 0$: point is on the n-th vertex lying on the face.

q_3: defined like q_1, which gives the position of the 2nd intersection point on the edge.

q_4: defined like q_2, which gives the position of the 2nd intersection point on the face.

q_5: gives the number of intersection points

$q_5 = 0$: no intersection

$= 1$: one intersection

= 2: two intersections

= 3: the edge lies on the face

q_6: q_6 = 1 if the direction of the edge at the first intersection point is the same as the direction of the normal vector of the face at the first intersection point

= -1 if it is in opposite direction

q_7: defined as q_6 for the second intersection point

These seven integers would be used to process intersection points as in merging.

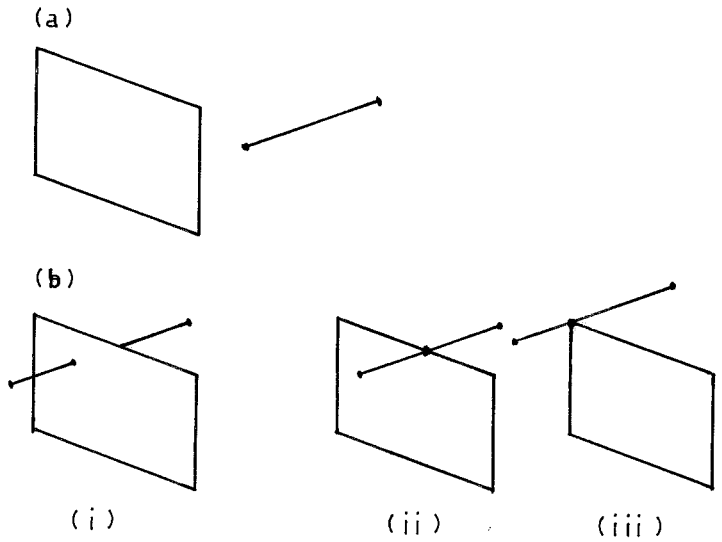
5.2.2.2. Processing Intersections

Fig. 5.2 summarises the various cases and their actions taken to process intersections of face-edge pairs.

Similar to merging, the processing of intersections is based on the one-intersection case. In the case of two intersections, the edge is first split at the first intersection point as if there were only one intersection and the resulting piece which contains the second intersection point is split again at that point.

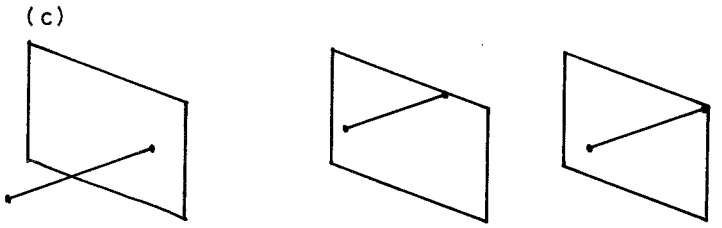
The procedure, in the case of one intersection point, is as follows:

1. If $q_1 \neq 1$ go to step 6
2. If $q_2 \neq 0$ go to step 5
3. Here $q_1 = 1$ and $q_2 = 0$.
   Create a new vertex at the first intersection point.
4. Split the edge e at the first intersection point. Let e_1, e_2 be the 2 pieces. Go to step 10.
5. Here $q_1 = 1$ and $q_2 = n > 0$, replace the first intersection point by the n-th vertex and go to step 4.
6. Replace the intersection point by the end vertex, indicated by q_1.
   If $q_2 = 0$ go to step 8.
7. $q_2 = n > 0$, replace either one, whichever is higher numbered, of the end vertex and the n-th vertex with the other and mark the retained vertex as special. Go to step 9.

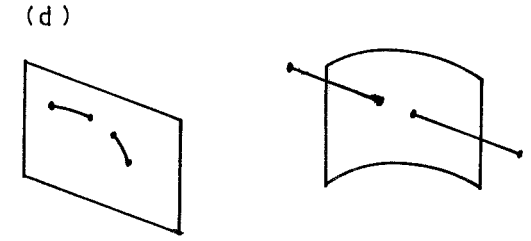


No intersection
No action taken

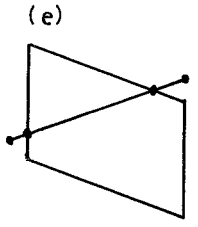
One intersection.
Split the edge at a new vertex ((i) and (ii)) or at an old vertex (iii). Mark the vertex as special. Record in the IVEF list.



One intersection.
Mark the end vertex as special. Record in the IVEF list.



Two intersections.
Split the edge as required. Record in the IVEF list and mark the intersection points as special.



The edge lies on the face. No actions taken.

Fig. 5.2 Types of Face-Edge Intersections and Actions taken

8. Mark the intersection point as special.

9. Set $e_1 = e_2 = 0$.

   If $q_1 = 3$ set $e_1 = e$.

   If $q_1 = 2$ set $e_2 = e$.

10. Record the intersection in the IVEF list.

    Set n_1 = intersection point.

    If $q_6 = -1$ set $n_2 = e_2$

    If $q_6 = 1$ set $n_2 = e_1$

    Set n_3 = face number

    Set n_4 = original number of the edge e.

5.3 The Third Stage

In merging, the third stage creates new faces by joining edges and/or pieces of edges together to make new faces. Here an edge of a new face is always an original edge or a piece of an original edge of the original face. The only new features which can be created are the vertices.

In intersection, new faces are created by joining not only the edges or pieces of the edges but also the new vertices. When new vertices are joined, new edges are created. In other words, although new faces always lie on the original faces, they can have new edges and vertices. New vertices are the intersection points of edges of one object and faces of the other while new edges are the intersection lines of faces of one object and those of the other.

Faces of the first object are assembled first. Some may be deleted. Some may remain unchanged. Some may be deleted and replaced by new faces on which new edges may be created and stored. Next, faces of the second object are assembled and as before, some may be completely deleted, some may remain unchanged and some may be replaced by new faces. This time no new edges are created because they have already been created and stored.

The general procedure is:

1. Get the next face of the first object. If there are no more faces, go to step 5.
2. If the face has been marked as non-intersecting, leave it as it is and go to step 1.
3. If the face has been marked as coplanar with another face, go to step 1.
4. The current face is intersecting and non-coplanar with any other faces. Call a routine to assemble the face (Section 5.3.1).
   Go to step 1.
5. Get the next face of the second object. If there are no more faces, go to step 10.
6. For the current face, make two lists, IVEF1 and IVEF2, which are extracted from the general IVEF list such that:
   (a) IVEF 1: contains all the entries in IVEF which have the face number (n_3) equal to the face number of the current face.
   (b) IVEF 2: contains all the entries in IVEF, which have the original edge number (n_4) equal to the edge number of any of the original edges of the current face.
7. If both IVEF1 and IVEF2 are empty, go to step 8.
   Otherwise go to step 9.
8. If the current face is marked as coplanar with other faces, go to step 9.
   Otherwise, the current face is non-intersecting and non-coplanar.
   Mark it as non-intersecting and go to step 5.
9. Call a routine to assemble the current face and the faces coplanar with it (Section 5.3.2).
   Go to step 5.
10. For each remaining face which is still marked as non-intersecting, determine if it is retained or deleted in the final object.
    (Section 5.3.3.)

11. The assembly process is complete.

    Unmark all faces and edges which are still marked as non-intersecting.

    Translate the object back to its original position.

    The third stage is complete.

In step 3, when a face F1 of the first object is found coplanar with a face of the second object, it is skipped. In step 9, this face F1 is assembled together with the face coplanar with it.

At the end of the assembly process, faces and edges which are still marked as non-intersecting are unmarked. This step makes the data structure compatible with that of an object at the end of the third stage in merging. This enables the fourth stage in merging, a "garbage-collecting" process, to be equally applied here to finish off the final object. Also because the object was first translated in stage 1, it is now translated back to its original position.

5.3.1 Assembling a face of the first object

The assembly routine is:

1. Make up the IVEF1 and IVEF2 lists for the current face of the first object.
2. If both IVEF1 and IVEF2 are empty, go to step 3.

   Otherwise go to step 4.
3. The current face is non-intersecting. It must be either wholly inside the second object, in which case it has to be deleted, or wholly outside, in which case it is retained.

   Get any one vertex V on the current face. Call a routine, TESTPO, to test if V is inside the second object (Section 5.3.1.1.) If V is inside, mark the current face as deleted and exit from the routine. If V is outside, leave the current face as it is and exit from the routine.
4. Make a simple edge list EL1 for the current face. (Simple edge

list EL1 is exactly like the simple edge list ELN described in merging).

Calculate SUM1, the area traced out by the perimeter of the current face as projected on the projection plane.

5. If the perimeter of the current face has been used, go to step 43.

6. If the perimeter of the current face has no intersections, go to step 40.

7. The perimeter of the face has intersections and has not been used. Make it the current loop. Start a new perimeter.

8. Get all the edges on the current loop.

9. Get the next edge E on the current loop.

If the edge has been used, go to step 39.

Let KV be the starting vertex of the edge.

If KV is a special vertex, go to step 12.

10. KV is not a special vertex. KV must be either inside or outside the second object. Use TESTPO to find this out. If KV is outside go to step 13.

11. Mark the edge E as used and go to step 39.

12. KV is a special vertex.

Look up the IVEF2 list for the entry that contains KV and get the retained edge in that entry. If the retained edge is the current edge E, go to step 13.

If the retained edge is not the current edge E, go to step 11.

13. If a hole is being made go to step 14.

Here, a perimeter is being made. Make an entry to the ENEW list, a temporary list storing new loops of the new face.

14. Initialize the new loop: set the initial vertex of the loop to be KV and the area test to be SUM = 0.

15. Add the edge E to the new loop.

16. Observe the end vertex V of the edge E.

If V is the initial vertex of the current loop, go to step 33.

If V is a special vertex, go to step 18.

17. V is not a special vertex. Make the next edge on the current loop (of the original face), which starts at this vertex the current edge E and go to step 15.

18. Look up the IVEF2 for an entry which contains the special vertex V. If no such an entry can be found, treat V as non-special and go to step 17.

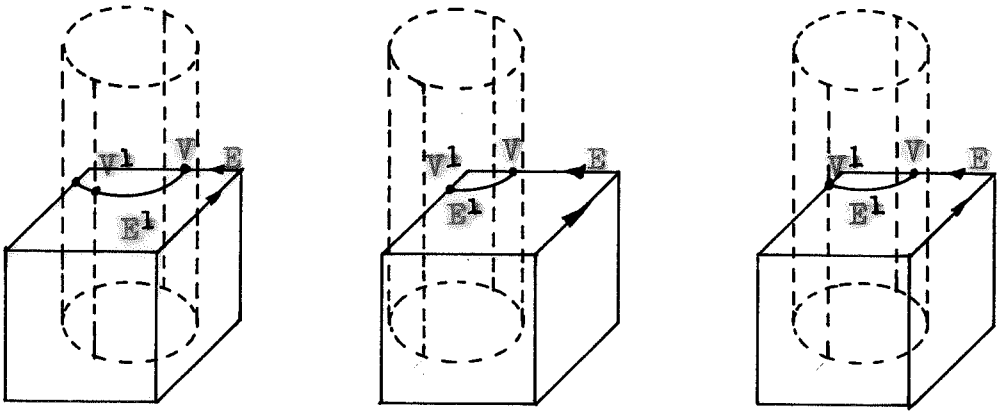
If one is found, let MF be the face in that entry. MF is the face of the second object which meets the current face at V. Therefore, the next edge to be added to the current new loop must be the one lying on the line of intersection of the current face and the face MF. That edge starts at V and must end at another special vertex V^1. V^1 can be on one edge of either face or it can be on two edges of the two faces (Fig. 5.3).

19. Initialize IVL list, a temporary list which contains all possible vertices, such as V^1.

20. Look up the IVEF2 list for an entry which has a face number equal to MF and a vertex number <u>not</u> equal to the current vertex V. If such entries are found, store their vertices in the IVL list. These vertices are the intersections of edges of the first object face and the face MF. (Fig. 5.3 (ii) and (iii)).

21. Call a routine to get all vertices on the face MF and store them in a separate list, say MVL list.

22. Look up the IVEF1 list for entries which have vertices not equal to the current vertex V, but are also present in the MVL list. If such entries are found, enter their vertices in the IVL list. These vertices are the intersections between the edges of the face MF and the current face. (Fig. 5.3.(i) and (iii)).



(i) V^1 on a second object edge

(ii) V^1 on a first object edge

(iii) V^1 on both edges of the two objects

The top face of the first object is being assembled.

_____ : the first object

\- - - - - - - - - : the second object.

E: the current edge of the new loop.

E^1: a new edge to be created and the next edge to be added to the new loop

V: the current special vertex

V^1: the next special vertex to be looked for.

Fig. 5.3 Creating new edges

23. All possible vertices V^1 are now in the IVL list. If IVL is empty, i.e. no such V^1 can be found, go to step 24.

If there is only one entry, make it V^1 and go to step 25.

If there is more than one entry, choose the vertex nearest to V, make it V^1 and go to step 25.

24. If a perimeter is being made, ignore V as special and go to step 17.

If a hole is being made, go to step 31.

25. If both the current face and the face MF are flat, make a new straight edge which starts from V and ends at V^1. Store the new edge in the straight edge list and go to step 27.

26. If either one of the current face and the face MF is curved, create

a new curved edge, starting at V and ending at V^1. Calculate the L-numbers and parameter limits for the new edge and store it in the curved edge list. However, the new edge may still be a straight edge, a fact which will be revealed immediately when the calculation of the L-numbers fails. In this case, create a new straight edge as in step 25.

27. Add the new edge to the current new loop being made.

Update the area traced by the current loop.

Store the new edge in a separate new edge list called NE. Each entry in the new edge list has 4 numbers:

W_1: vertex number of V^1

W_2: edge type

W_3: face number of face MF

W_4: edge number of the new edge

28. If V^1 is the initial vertex of the current new loop, go to step 33.

29. Assign V^1 as the current vertex V.

30. If the current vertex V was on an edge of the first object, i.e. found in step 20, the next edge to be added to the new loop would be from the current face. Treat V as non-special and go to step 17.

Otherwise the current V was on an edge of the face MF.

The next edge to be added to the new loop would be the intersection of the current face and another face of the second object which meets the face MF at the edge on which V was. Look up the IVEF1 list again for the original edge on which V was. Call it current edge E.

31. Find the face MF1 of the second object which meets the face MF at the edge E.

Assign MF1 as the face MF.

Initialize the IVL list.

32. If a perimeter is being made, go to step 20.

    If a hole is being made, go to step 21.

33. The initial vertex is met i.e. the new loop closes.

    If the new loop is a hole, go to step 36.

34. Check the area of the new loop is non-zero.

    If it is zero, go to step 39.

35. The new loop is the perimeter of a new face. If there are any holes in the original face, find those which have not been used and which are shown to be inside the new perimeter and add them to the new face.

    Go to step 37.

36. The new loop is a hole of the new face. If SUM * SUM1 > 0, i.e. if the area of the new loop is of the same sign as the area of the perimeter of the current face, the new hole has been traced in the wrong direction. Reverse the direction of the hole and its edges. Add this hole to the new face. This hole is the last hole of the new face.

37. Add the new face to the data structure. The new face has the same face type as the original face.

38. If the loop just made is a hole, go to step 5.

    If it is a perimeter, go to step 39.

39. If there are more un-used edges on the current loop of the original face, go to step 9.

    If there are no more edges left, go to step 43.

40. The perimeter of the current face has no intersection. Initialize the ENEW list and copy the perimeter to it as the new perimeter of a new face.

41. Add all holes of the original face, which have no intersections and are shown to be outside the second object, to the new face.

If there are un-used holes with intersections, find one and make it the current loop of the original face and go to step 8.

42. There might still be a final hole created by the second object cutting through the current face without cutting any edges on the current face.

Set MF = 0 and start a new hole.

Set the initial vertex of the hole to be the vertex in the first entry of the IVEF1 list. Set V to be this vertex as well. The edge in this entry is the edge E.

Set the area test SUM = 0.

Go to step 31.

43. The assembly is complete.

Mark the original face as deleted and exit from the routine.

In Braid's work, the method of finding all possible vertices V^1 was different. He suggested that two separate lists similar to IVEF1 and IVEF2 be made for the face MF. Then the equation of the intersection line between the current face and the face MF must be found and its direction at the starting vertex V must also be calculated. Then each vertex in the IVEF1 and IVEF2 of MF is tested to see if it lies on the intersection line and in the right direction. Again, store all vertices that do, in the IVL list. This method is based on the actual geometry of the two objects. It has one advantage: when a loop is formed, it is always traced in the right direction. However, it requires more storage space for the two extra lists and more calculations. The approach given here from steps 19 through to 22 is based on the data structure without any consideration to the geometry of the two objects. Every point that lies on the current face is looked up within the data structure to see if it also lies on a face of the second object. All those that do must also lie on the inter-

section line and the vertex nearest to the starting vertex in either direction is chosen. Therefore a completed loop might have been traced in the wrong direction and an area test in step 36 has to be used. If this is the case, the loop is simply reversed.

Another point which should be noted is the way new edges are stored in the NE list (step 27). The NE list will be used again later to assemble faces of the second object. Now, every edge is the intersection line of two faces and is traced twice, once on each face, in opposite directions. Since a new edge goes from V to V^1 on the current face, it must go from V^1 to V on the face MF. Therefore, the entry holds V^1 as the starting vertex (W_1) and MF as the face number (W_3).

5.3.1.1. Subroutine TESTPO

Subroutine TESTPO determines whether a given point lies inside or outside an object when a point is known to be inside the object.

This problem arises in step 3 when it is necessary to know whether a point P lies inside or outside the second object.

Braid suggested that a straight ray is passed from the point P. Its intersection points with the planes of the faces of the object are noted. For each intersection point, INSPEC or CINSPE must be used to discover further if the point actually lies inside the perimeter of the corresponding face. Count the number of intersection points that do. If it is odd, P is inside the object. If it is even, P is outside.

Here, the fact that the origin O of the coordinate system is known to be inside the second object can be used to greatly simplify the problem.

Suppose the object has only flat faces.

Let the equation of a flat face be:

$$f(x, y, z) = Ax + By + Cz + D = 0.$$

The flat face divides the space into two half spaces. A point (x_1, y_1, z_1) in one half space yields $f(x_1, y_1, z_1)$ which is of opposite

sign to $f(x_2, y_2, z_2)$ yielded by another point (x_2, y_2, z_2) in the other half space. Now, for the origin O, which is inside the object:

$$f\ (0, 0, 0) = D$$

If the point P (x_p, y_p, z_p) is inside the object then the product $f\ (x_p, y_p, z_p)$. $D > 0$ for every flat face of the object. If the point P is outside, there must be at least one face equation which gives the product $f\ (x_p, y_p, z_p)$. $D < 0$.

If the object has curved faces, a similar approach is used. A convenient point A on the curved face is chosen (e.g. a point corresponding to $s = t = 0.5$). The normal vector $\bar{n}$ to the curved face at A is calculated. Then the product to be observed now is $(\overline{AP}.\bar{n})(\overline{AO}.\bar{n})$ instead of $f\ (x_p, y_p, z_p)$. D.

The algorithm thus proceeds as follows:

1. Get the next face of the second object. If there are no more faces, go to step 8.
2. If the face is curved, go to step 6.
3. The face is flat, Calculate the product

   $PD = f\ (x_p, y_p, z_p)$. D.
4. If the product $PD > 0$, go to step 1.

   If the product $PD < 0$, go to step 5.
5. The point P is outside the object.

   Exit from the routine.
6. Calculate the normal $\bar{n}$ of the curved face at the point

   $A(s = 0.5, t = 0.5)$.
7. Calculate the product $PD = (\overline{AP}.\bar{n})(\overline{AO}.\bar{n})$. Go to step 4.
8. Point P is inside the object.

   Exit from the routine.

5.3.2 Assembling a face of the second object

In assembling faces of the second object, no new edges are made because they have already been created in the previous stage. Their existence is stored and noted in the NE list. Edges, edge pieces and new edges on the face of the second object and those on its coplanar faces are joined up to form loops. An area test will show which loops are perimeters and which loops are holes. The holes are tested and assigned to their corresponding perimeters. Each perimeter together with its assigned holes makes up a new face.

The procedure is:

1. Initialize NEWL list, a list holding all loops to be created.
2. If both IVEF1 and IVEF2 for the currect face of the second object are both empty, go to step 4.
3. Search in the new edge list NE for those lying on the current face. Store them in a separate list, IVEF3.

   Each entry in the IVEF3 has 3 numbers:

   i_1: starting vertex of the new edge on the current face

   i_2: type of the new edge

   i_3: edge number
4. Make the simple edge list EL1 for the current face. Calculate the area SUM1 of the perimeter of the face. SUM1 has the sign of the area of the new perimeter of the new face.
5. If the current face is not coplanar with any other faces, go to step 31.
6. The current face is coplanar with other faces.

   Delete those coplanar faces which are found to be inside the current face and unmark those which are found to be disjoint with the current face.
7. If there are no more faces which are still marked as coplanar

with the current face, mark the current face as non-intersecting and exit from the routine.

8. For each remained coplanar face, make a simple edge list EL2. Calculate the area SUM1 of the perimeter of EL2. SUM1 has the sign of the area of the new perimeter of the new face.

9. If the perimeter of EL2 has intersections, go to step 14.

10. The perimeter of EL2 has no intersections. It must enclose the current face. Make this perimeter a new perimeter. Store it in the NEWL list and record its area.

11. If the perimeter of the current face has intersections, go to step 13.

12. The perimeter of the current face has no intersections. Add it as a hole in the NEWL and record its area. The remaining holes of the coplanar face which are found to be outside the current face are also added to the NEWL list and their areas are kept. This step is done by going to step 13.

13. Make the first hole in the EL2 list the current loop and go to step 16.

14. Make the perimeter of the EL2 list the current loop and go to step 16.

15. Make the next loop in the EL2 list the current loop. If there are no more loops, go to step 32.

16. Get all the edges of the current loop and make the first edge the current edge.

17. If the current edge has been used, go to step 30.

18. Get the starting vertex of the current edge. If it lies inside or on the second object, go to step 30.

19. The vertex lies outside the second object. Start a new loop in the NEWL list. Set the area test SUM = 0. Set the current vertex the initial vertex of the new loop. Also treat it as if

it were special.

20. If there are no coplanar faces, go to step 26.

21. Look up the EL2 list for an edge which starts at the special vertex and make it the current edge.

22. Add the current edge to the new loop.
Update the area traced by adding this edge.
Observe the end vertex of the current edge.

23. If the end vertex is the initial vertex of the new loop being made, go to step 29.

24. If the end vertex is not special, make the next edge on the current, original loop, which starts at the end vertex the current edge. Go to step 22.

25. If the end vertex is special, go to step 26.

26. Look up the EL1 list for an edge which starts at the special vertex and add it to the new loop. Update the area of the new loop. Keep adding edges as before until:
- the initial vertex is met, go to step 29.
- a special vertex is met, go to step 27.

27. Look up the new edge list IVEF3 for an un-used edge which starts at the special vertex. Make it the current edge. If no such edge can be found, go to step 20.

28. Add the current edge to the new loop. Update the area of the loop. Observe the end vertex of the current edge (this end vertex is special). If it is the initial vertex, go to step 29. If it is not, go to step 27.

29. The new loop closes. If the product SUM.SUM1 > 0, the new loop is a perimeter. If the product is negative, the new loop is a hole. Store it appropriately and record its area.

30. Make the next edge of the current original loop the current edge

and go to step 17.

If there are no more edges, go to step 31.

31. Look up the new edge list IVEF3 for an un-used edge. Make it the current edge and start a new loop with its starting vertex as the initial vertex. Reset the area SUM of the new loop. Go to step 28. If all new edges have been used, go to step 15.

32. All possible perimeters and holes have now been created. If there are no holes, go to step 38.

33. Make the largest, un-used hole the current hole. If all holes have been used, go to step 38.

34. Find all un-used perimeters which are larger than the current hole. Among those, make the one with the least area the current perimeter.

35. If the current hole is inside the current perimeter, go to step 37.

36. The hole is outside the perimeter. Among the perimeters found in step 34, get the next larger perimeter. Make it the current perimeter and go to step 35.

37. Assign the current hole as a hole of the current perimeter. Decrease the area of the perimeter by the area of the hole. Find the next largest hole which can fit into this decreased area of the perimeter, make it the current hole and go to step 35. If no such a hole can be found, it means that all holes which are in the current perimeter have been so assigned. Go to step 33.

38. Get the next perimeter and start a new face. If there are no more perimeters, go to step 41.

39. Add the current perimeter to the new face. Add all holes, if any, which have been assigned to this perimeter, to the new face.

40. Add the new face to the data structure. Go to step 38.

41. The assembly is complete.

Mark the current face and its coplanar faces as deleted.

Exit from the routine.

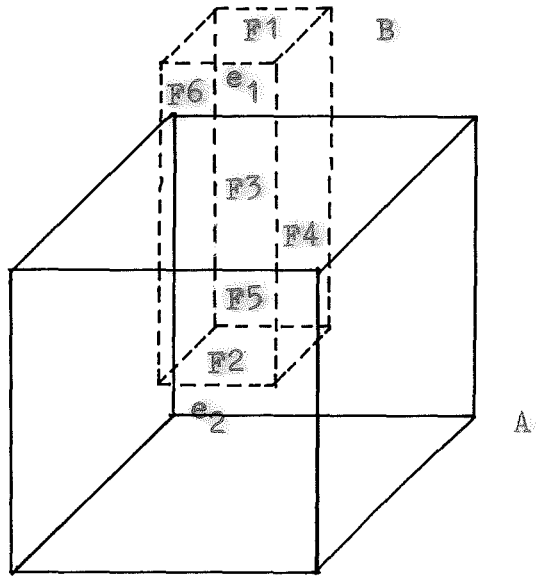
Note that when the current face is not coplanar with other faces, the area sign of its perimeter is the area sign of the perimeter of the new face. However, when it is coplanar with another face, the area sign of the perimeter of the coplanar face is the area sign of the perimeter of the new face.

5.3.3 Processing Non-Intersecting Faces of the Second Object

Non-intersecting faces of the second object are either inside or outside the first object. From the geometrical point of view, two cases exist:

1. If the first object is positive, then those non-intersecting faces which are shown to be inside the first object are retained and those which are shown to be outside are deleted.
2. If the first object is negative, the reverse action is taken, i.e. delete those inside the first object and retain those outside. (Fig. 5.4).

A different approach is used here, based on the data structure and not on the geometry of the object. The basis for this approach is the fact that every edge is the intersection between two faces and if that edge has previously been used to reassemble another face, the non-intersecting face which contains that edge will be retained. Fig. 5.4(i) shows two non-intersecting faces F1 and F2 and one intersecting face F3 which meets F1 and F2 at edges e_1 and e_2, respectively. If A is positive then when the face F3 is re-assembled, edge e_2 is used as part of the perimeter of the new face F3 while edge e_1 is discarded. The same is true for every other edge of F1 and F2. Since edge e_2 of F2 has been used to re-assemble F3, F2 is retained. Similarly since edge e_1 of face F1 has been discarded, F1 is deleted. The reverse action applies to the case when A is negative. This approach has the

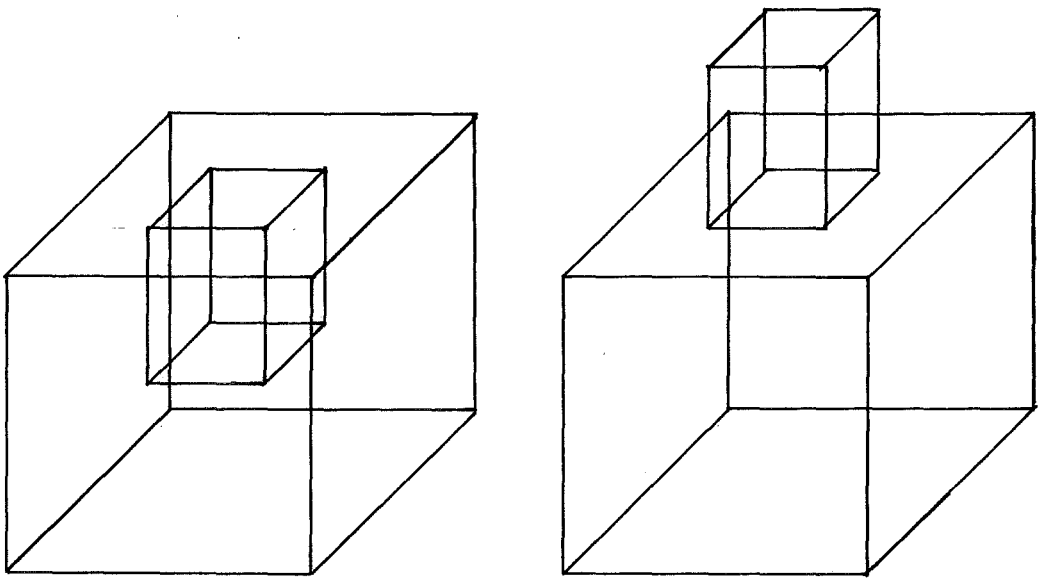


(i)

A : the first object

B : the second object

F1 & F2 : non-intersecting faces of B



(ii)

A is positive

F1 is deleted

F2 is retained

(iii)

A is negative

F1 is retained

F2 is deleted

FIG.5.4. Processing non-intersecting faces of the second object.

advantage that no distinction has to be made to whether A is positive or negative. It already did implicitly when intersecting faces are assembled.

The processing procedure is:

1. Get the next non-intersecting face of the second object and make it the current face. If there are no more faces, go to step 6.
2. Get all the edges of the current face.
3. Get the next edge of the current face. If there are no more edges, go to step 5.
4. Check among the other, undeleted faces of the second object to see if any one of them contains the current edge. If one of them does, unmark the current face and retain it. Then go to step 1.
   If the current edge has not been used, go to step 3.
5. All edges of the current face have not been used.
   If this is the first pass, leave the face as it is and go to step 1.
   If this is the second pass, mark the face as deleted and to go step 1.
6. If this is the first pass, go back to the beginning of the face list and run through the process a second time starting from step 1. If it is the second pass, exit from the routine.

Two passes are necessary because sometimes a non-intersecting face neighbours only other non-intersecting faces (Fig. 5.5). The first pass would decide on the existence of those non-intersecting faces which meet other intersecting faces (e.g. faces F1, F2, F3, F4). The second pass would decide on the existence of those non-intersecting faces which do not meet any intersecting faces (e.g. face F5). If the second object is complex enough three or more passes may be necessary. However, for this application, two passes were found to be sufficient.

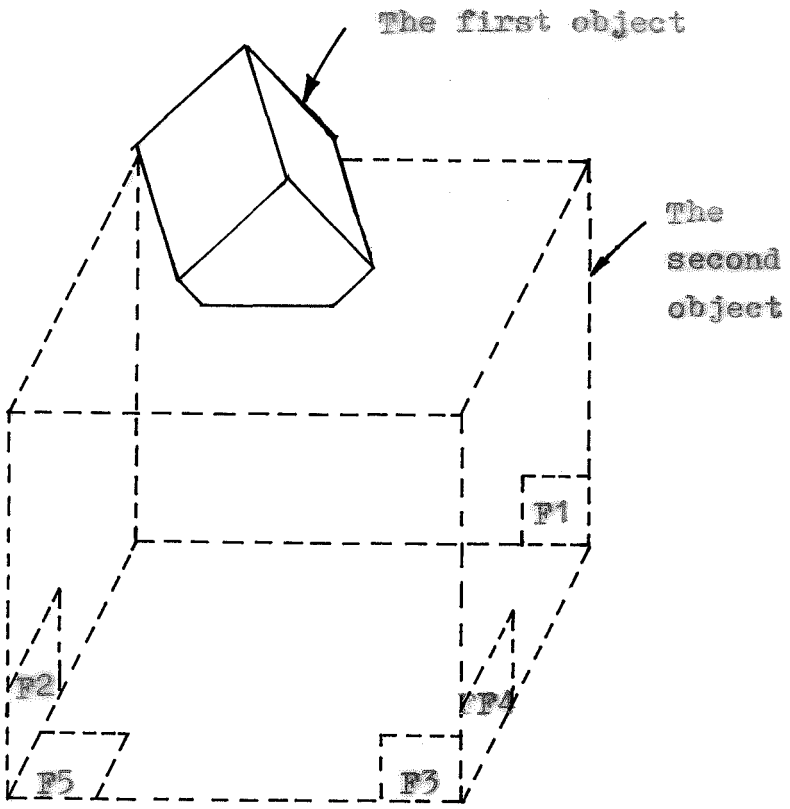


Fig. 5.5. The non-intersecting face F5 of the second object only meets other non-intersecting faces (F1, F2, F3, F4)

5.4 The Fourth Stage

The fourth stage is exactly the same as the fourth stage in merging.

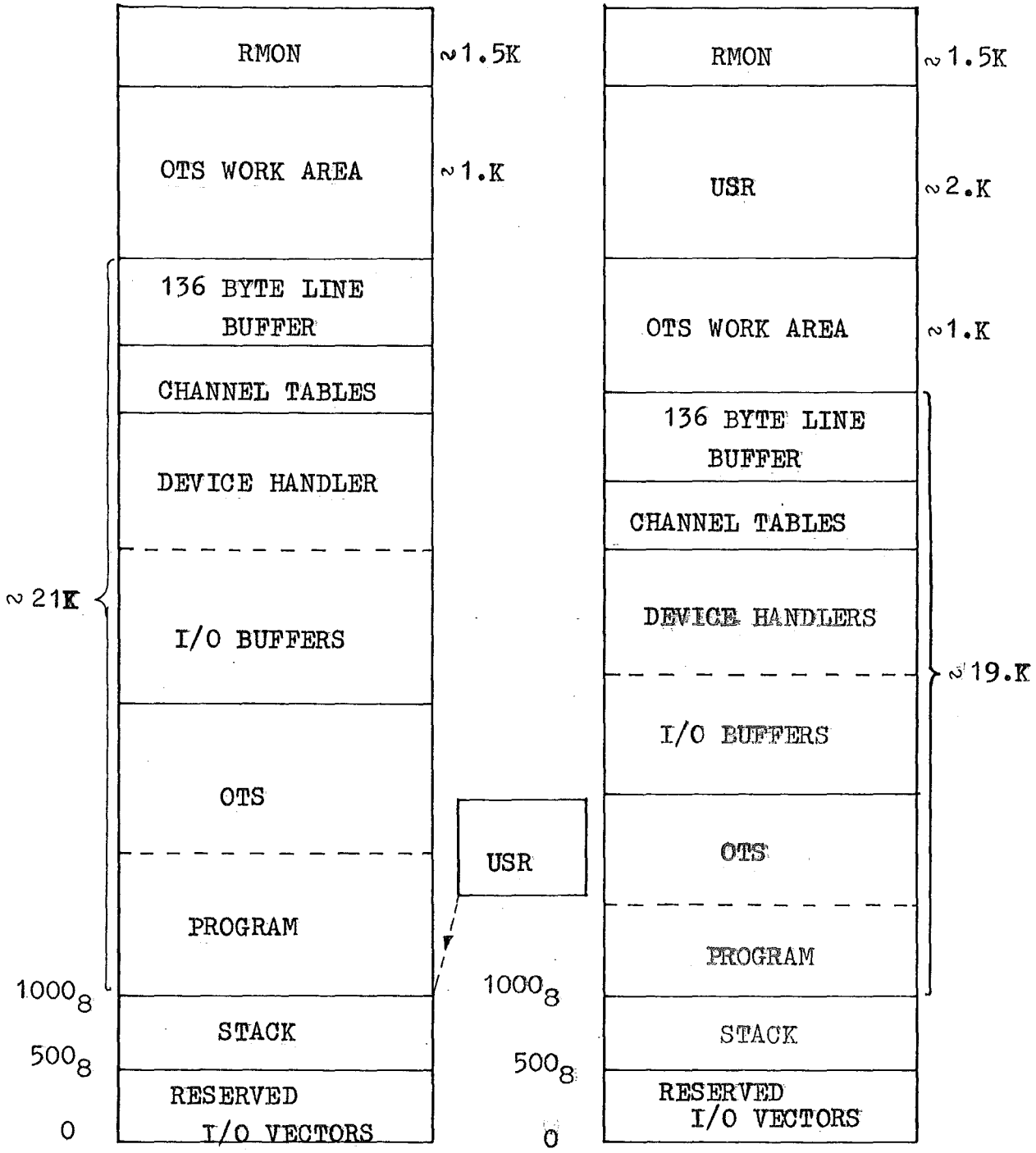
CHAPTER 6

THE SYSTEM PROGRAMS

6.1 The System Restriction

Although the PDP-11/40 computer has 24K words of memory in core, the actual amount of core available to a FORTRAN user is much less than this. Fig. 6.1 shows the memory organisation in the core when a Fortran program is being run [20]. Some system components are always resident in core and are of fixed sizes. These include RMON, OTS work area, the stack, the reserved I/O Vector (interrupt vector) area and in the USR (User's Service Routines) resident system, the USR area. Other components may vary in sizes depending on the particular program and some may not be resident at all. In the USR swapping system, the USR is not locked in core. When the USR is required, it is copied from the system device (disk) onto core, just on top of the stack area. After it has completed the required task, the area is returned to the program again. By doing this, the system offers more core to the user's program. However, when the graphics system is used as well, the swapping of the USR undesirably overlays the display codes and stops the display handler. Therefore in this application, the USR resident system must be used and this reduces the amount of core available to the user by 2K words. Besides, other system components require a fair proportion of the remaining core so that the actual amount of core available to the user's program is considerably less. Because of this restriction, the program has to be overlaid, a common arrangement for a program run in small computers and especially in an interactive mode.

Fig. 6.2 shows the memory allocation map during runtime of a Fortran program which uses the graphics system and is in an overlay structure. In this configuration, the 136 byte line buffer and the



(a) Swapping System (b) Resident System

RMON : Resident Monitor

OTS : Object - Time System

USR : User's Service Routines

Fig. 6.1. Runtime Memory Organization.

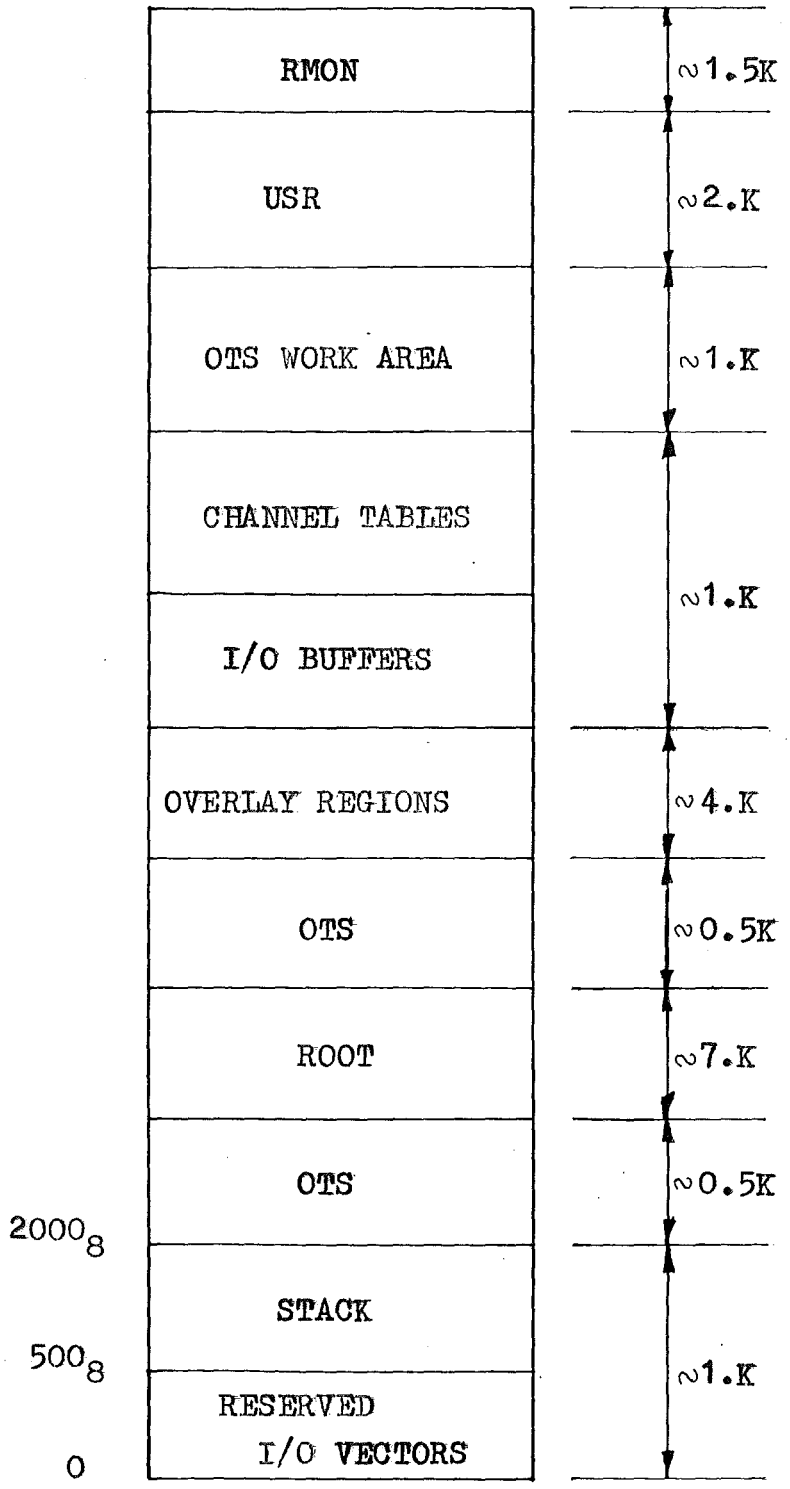


Fig. 6.2. A Typical Runtime Memory Oganization in The RT-11 24K System.

device handlers area are omitted because the system does not have a line printer and handles no other devices than its system disks. Fortran programs use a lot of pointers and it has been found that the area reserved for the stack by default (from 500_8 to 1000_8) is insufficient. Therefore the stack area has been expanded to be from 500_8 to 2000_8. This can be done by using switch /B:2000 when the object programs are linked. Experience shows that in this application, the channel table and the I/O buffer areas occupy approximately 1K words of memory while the OTS occupies about 6K words in the area between the root of the program and the overlay regions and other 0.5K words in the area between the root and the stack. This leaves an overall area of approximately 11K words for the user's program. However, in order to display a reasonable picture, a sizeable area has to be reserved for the display buffer. This area is approximately 3K words. Also a data area (or a common block) needs to be about another 3K words to hold the data structure of a reasonable object. The display buffer and the data area has to come from the root area. Hence the actual Fortran program has about 1K words for the root and 4K words for its overlay regions. Because of this severe restriction, it is impossible to load the complete graphics system into core as one program even with the use of overlays. The system has had to be divided into many small programs. Each program does a specific job and can be run as often as it is required. The intermediate data between programs is saved on a disk file. The rest of this chapter describes what the programs are, what they do and the structure of the disk file. Appendix C describes the programs in greater detail.

6.2 The Programs

1. TGDEF:

This program should always be the first to be run. The program defines, creates all primitive objects. All primitive objects when

created are positive and they can be negated on request. Primitive objects, once created, can be deleted and re-defined. Any one object can also be turned "off" or "on" from the display screen. Their data structures are stored on the disk file which can be accessed by any other program.

Primitive objects created by this program will be added or subtracted among themselves to build the required product shape.

2. Merging: Batch Job TGMERG

The program written for the merging algorithm was found to be too big for the available amount of core and therefore it had to be subdivided into five separate programs. They are:

(a) TGMER1: This program does the first stage of merging as described in chapter 4.

(b) TGME21: This program does part of the second stage of merging. It discovers every pair of coplanar faces and for each pair it reveals and stores all intersections between straight edges.

(c) TGME22: This program does the remaining part of the second stage. Namely it deals with all intersections involving curved edges.

(d) TGMER3: Does the third stage of merging.

(e) TGMER4: Does the fourth stage of merging.

Complete merging of two objects will be done by running TGMER1, TGME21, TGME22, TGMER3, and TGMER4 consecutively and in that order. At the end of each sub-program the intermediate data structure of the object is written on a reserved-area in the data file on the disc. This data structure becomes the input file to the next sub-program at the beginning of its execution.

Because these sub-programs are run one after the other in a strict order and require no interaction from the user, they can be

run "unattended" in a batch stream by the RT-11 BATCH. (19) Briefly, BATCH is a job control language that enables programs to be run with little or no interaction with the user. The batch stream is in a file called TGMERG. It consists of five consecutive commands, each of which runs one sub-program. When TGMERG is run under BATCH, each command in the batch stream is executed in the written order. This facility allows the user to run only one program, namely TGMERG, instead of five programs.

3. Intersection: Batch Job TGINTN

For the same reason, the intersection algorithm had to be written into four separate programs. They are

(a) TGINT1: Does the first stage of the intersection algorithm as described by chapter 5.

(b) TGINT2: Does the second stage.

(c) TGINT3: Does the third stage.

(d) TGMER4: Does the fourth stage which is exactly the same as the fourth stage of merging as mentioned in chapter 5.

The complete intersection algorithm is again done by running TGINT1, TGINT2, TGINT3 and TGMER4 consecutively and in that order. This running sequence is again stored in a batch stream under a file name TGINTN. In other words, the intersection algorithm can be run by one single file, TGINTN, under the RT-11 BATCH.

The intermediate data structure of the object at the completion of each sub-program is stored in the same reserved area on the disc file.

4. TGDESN:

This is an interactive program which assists the user to design a general assembly of a compression die. The program follows exactly

the design procedure detailed in chapter 1. This program can access the data structures of previously-defined objects such as the product, the compression machines and standard die components. It can also create new objects on the user's request and store them on the disc file.

5. TGNEG:

This program negates a general object. The data structure of the negated object can be stored separately, thus retaining the original object or can be over-written on the original object, thus deleting the original object.

6. TGDIS:

This program displays any objects on the GT-44 screen and also builds up display files on request. The display files can then be used to produce drawings on the system plotter.

A typical design job can be done in the following procedure:

(1) Defining the primitive objects that constitute the product. This is done by running TGDEF.

(2) Merging and/or intersecting the primitive objects to build up the product. This is done by running TGMERG, TGINTN under BATCH.

(3) Building up the general die assembly by running TGDES.

(4) Extracting the individual die components from the assembly. This is again done by running TGMERG, TGINTN.

(5) Producing drawings of the generated objects by running TGDIS.

When running TGMERG, TGINTN the intermediate object may require negation. This is done by running TGNEG.

These programs could have been combined into one single system if more core had been available. This is simply done by inserting more commands in the root to control the execution of the individual routines.

6.3 The Data Disc File

Data structures of objects are stored in a direct-access file on the disc. A direct-access file is defined by the number of records NR in the file and the record size IS (the number of words per record). Data transfer to and from the file takes place in one whole record at a time and therefore the I/O buffer for that file has the same size as the record size.

One would suggest that the record size be big enough to contain a complete data structure of one object so that any object can simply be fetched or saved by a single I/O operation. However, this record size has a serious disadvantage. Since objects involved in this application vary in sizes, the record size must be big enough to accommodate the biggest object and a big record size means a big I/O buffer. In the situation where the available amount of core is greatly limited, a file of that record size is unacceptable.

Under the PDP-11 hardware, an integer is represented by one word and a real number is represented by two words. This means that the smallest possible record size is two words. In other words, each record can contain one real number or two integers. The number of records required to hold the complete data structure of an object (a data block) can be precisely calculated from its 8-integer header: the number of vertices, the number of faces, the number of flat faces, the number of straight edges, the number of curved edges, the edge list length, the number of groups of L-numbers and the number of parameter limit pairs (Appendix B). These 8 integers form the header of each data block and they occupy the first four records of the data block.

Also because the maximum value an integer can be represented is 32766, the file is conveniently defined to have 32760 records. Fig. 6.3 shows the organisation of the disc file. The object header

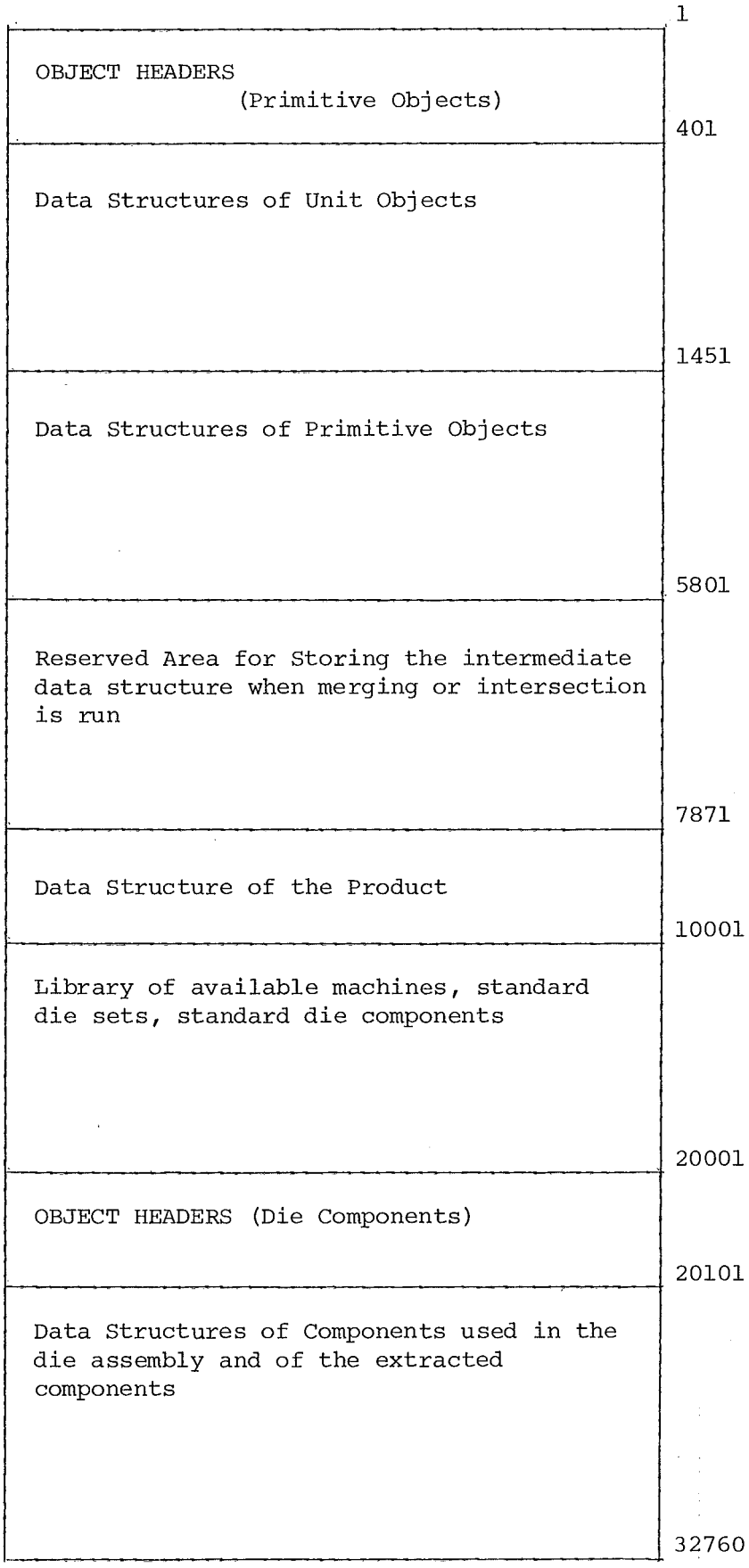


Fig. 6.3. The Organisation of the Data File.

for each object is a group of records containing the parameters defining the object and the record number of the start of the object's data block.

The first 400 records contain the object headers for the primitive objects whose data structures are held in the file (from record 1451-5801). The next 1050 records hold the data structures of the unit objects as described in appendix A. The next area (from record 1451-5801) holds the data structure of the primitive objects which have been generated. This area can hold a maximum of 20 primitive objects. The area between record 5801 - 7871 is reserved for holding the intermediate data structure between stages when merging or intersection is run. The data structure of the product shape is held starting at record 7871. Ten thousand records (10 001 - 20 001) are reserved for holding the library of the available machine and standard die components. The rest of the file holds the data structures of the individual die components generated as the design procedure proceeds.

CHAPTER 7

A DESIGN EXAMPLE

A very simple die design using the compression moulding process is given in this chapter. The product to be moulded is shown in Fig. 7.1. The product is in fact an electrical blank plate (CAT. NO. 250) manufactured by PDL Industries Ltd., Christchurch. The plate is simplified in its shape to reduce the unnecessary complexity which is not required for the purpose of this illustration. The material for the plate is Urea Formaldehyde, a thermoset. All dimensions are in millimetres.

7.1 Product Description

The product was observed and it was found that it could be described by adding and subtracting six primitive objects as shown in Fig. 7.2. It was formed by subtracting PYR2 from PYR1, then adding CON3 and CON4 and finally subtracting CYL5 and CYL6.

To define these primitive objects, TGDEF was run: a print-out is shown below. Note that the product was formed with its reference point R.P. at the origin of the co-ordinate system.

The data structures of these primitives were stored on a disc file. Also for each (positive) primitive, a negative object was generated and stored as object 7 through to object 12. This negation was necessary because some of the primitives were in fact holes in the object or would become holes in the die assembly.

The product was formed by merging negative PYR2 into positive PYR1, followed by merging positive CON3 and positive CON4 into the result and finally merging negative CYL5 and negative CYL6. The data structure of the product was again saved on the disc file. A computer printout of the first two mergings is shown below. Note

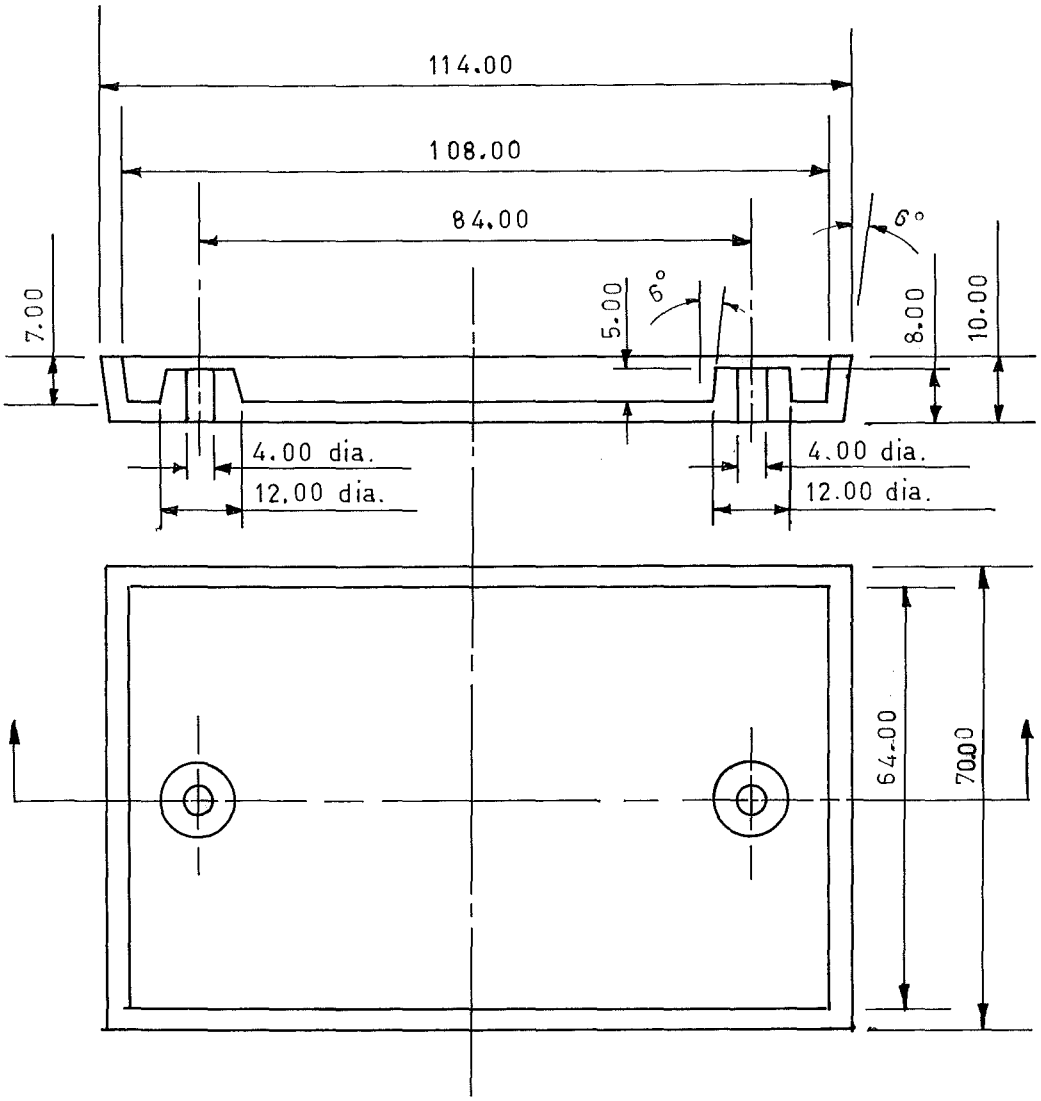


FIG.7.1. THE PRODUCT A BLANK PLATE

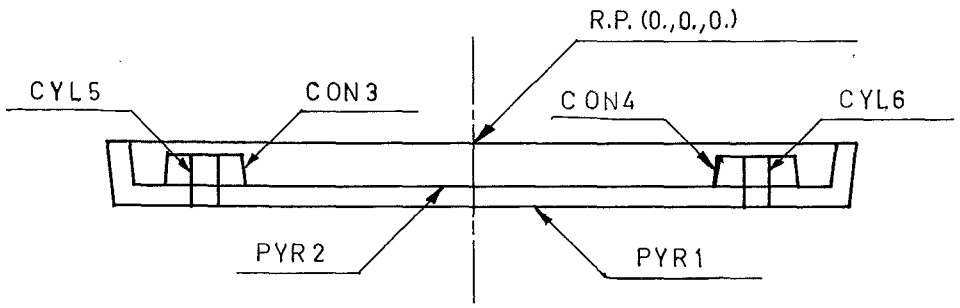


FIG.7.2. BUILDING UP THE PRODUCT

```
. RUN TDEF

ENTER SCALE PARMTR: 200

ENTER VIEW POINT IN 3F10.0
1,2,3

*
DEF

**
PYR

ENTER 3 DIM.S AND 2 TAPERED ANGLES IN 5F10.0,THEN RP IN 3F10.0,  THEN AXIS
114,70,10,6,6
0,0,0
0,-1,0

SUBP     1      IS A TPYR AT          0.00000          0.00000          0.00000
DIM.S ARE       114.00000         70.00000         10.00000
TAP.ANG.ARE         6.00000          6.00000
AXIS VECTOR IS          0.00000         -1.00000          0.00000
REC.  NO. = 1451
**
PYR

ENTER 3 DIM.S AND 2 TAPERED ANGLES IN 5F10.0,THEN RP IN 3F10.0,  THEN AXIS
100,64,7,6,6
0,0,0
0,-1,0

SUBP     2      IS A TPYR AT          0.00000          0.00000          0.00000
DIM.S ARE       100.00000         64.00000          7.00000
TAP.ANG.ARE         6.00000          6.00000
AXIS VECTOR IS          0.00000         -1.00000          0.00000
REC.  NO. = 1661
**
CON

ENTER RP,BASE DIA.,HEIGHT,BASE ANGLE IN 6F10.0
THEN AXIS VECTOR IN 3F10.0
-42,-7,0,12,5,84
0,1,0

SUBP     3      IS A TCONE AT       -42.00000         -7.00000          0.00000
BASE DIA=   12.00000      HEIGHT=    5.00000      BASE ANGLE=   84.00000
AXIS VECTOR IS          0.00000          1.00000          0.00000
REC.  NO. = 1871
**
CON

ENTER RP,BASE DIA.,HEIGHT,BASE ANGLE IN 6F10.0
THEN AXIS VECTOR IN 3F10.0
42,-7,0,12,5,84
0,1,0

SUBP     4      IS A TCONE AT         42.00000         -7.00000          0.00000
BASE DIA=   12.00000      HEIGHT=    5.00000      BASE ANGLE=   84.00000
AXIS VECTOR IS          0.00000          1.00000          0.00000
REC.  NO. = 2081
**
CYL
```

```
 ENTER CP, LENGTH, DIA.  IN 5F10.0
THEN AXIS VECTOR IN 3F10.0
-42,-6,0,8,4
0,1,0

SUBP     5      IS A CYL AT        -42.00000         -6.00000          0.00000
LENGTH=    8.00000            DIA=    4.00000
AXIS VECTOR IS          0.00000          1.00000          0.00000
REC.  NO.= 2291
**
CYL

 ENTER CP, LENGTH, DIA.  IN 5F10.0
THEN AXIS VECTOR IN 3F10.0
42,-6,0,8,4
0,1,0

SUBP     6      IS A CYL AT         42.00000         -6.00000          0.00000
LENGTH=    8.00000            DIA=    4.00000
AXIS VECTOR IS          0.00000          1.00000          0.00000
REC.  NO.= 2501
**


*
NEG

ENTER  OBJ.  NO.  TO BE NEGATED:1

OBJ.    7  IS NEG.  OF OBJ.    1
REC.  NO.= 2711
ENTER  OBJ.  NO.  TO BE NEGATED:2

OBJ.    8  IS NEG.  OF OBJ.    2
REC.  NO.= 2921
ENTER  OBJ.  NO.  TO BE NEGATED:3

OBJ.    9  IS NEG.  OF OBJ.    3
REC.  NO.= 3131
ENTER  OBJ.  NO.  TO BE NEGATED:4

OBJ.   10  IS NEG.  OF OBJ.    4
REC.  NO.= 3341
ENTER  OBJ.  NO.  TO BE NEGATED:5

OBJ.   11  IS NEG.  OF OBJ.    5
REC.  NO.= 3551
ENTER  OBJ.  NO.  TO BE NEGATED:6

OBJ.   12  IS NEG.  OF OBJ.    6
REC.  NO.= 3761
ENTER  OBJ.  NO.  TO BE NEGATED:

*
FIN

GOOD BYE
PAUSE --
TYPE <CR> TO EXIT
```

```
 ASS TT:LOG

.ASS TT:LST

.LOAD BA,TT

.R BATCH
*TGMERG/X


$JOB/RT11


ENTER REC. NOS. OF OBJ. 2 & OBJ. 1 :
?2921,1451

STOP --


STOP --


CURVED EDGE INTERSECTIONS ? :?

STOP --


STOP --


ENTER REC. NO. TO BE WRITTEN ON:?7871

STOP --


$EOJ

END BATCH

.R BATCH
*TGMERG/X


$JOB/RT11


ENTER REC. NOS. OF OBJ. 2 & OBJ. 1 :
?1871,7871

STOP --


STOP --
```

```
CURVED EDGE INTERSECTIONS ? :?

STOP --


STOP --


ENTER REC. NO. TO BE WRITTEN ON:??87J.

STOP --


$EOJ

END BATCH

. R BATCH
*TGMERG/X
```

(a) The Product after The First Two Mergings.

(b) The Complete Product.

Fig. 7.3.

that the first three commands were just to load the system RT-11 BATCH. Fig. 7.3.(a) shows the computer-generated picture of the product at this stage and Fig. 7.3.(b) shows the complete product.

7.2 The General Die Assembly

A program called TGDESN was written, based on the design process for a compression mould outlined in chapter 1.

This program was run to build up a general die assembly. The print-out is shown and details of the events and activities as they would occur are discussed below.

1. The first piece of information required by TGDESN is the scale parameter. This scale parameter defines the window screen area for the graphics display (Appendix C). In this case, the window is a square whose lower, left-hand corner has co-ordinates of (-250., -250.) and whose upper, right-hand corner has co-ordinates of (250., 250.).

2. The program chooses the XY plane as the projection plane. This plane will give the elevation of the die assembly.
   Then the starting record of the product's data structure is asked for. The user enters 7871. The computer then fetches the data structure from the disc file, starting at this record and displays the product on the projection plane as shown in Fig. 7.4(1). The position of the product is as it is, i.e. with its reference point RP at the origin. The computer also wants to know what primitive objects constitute the product and their negatives because these primitives will later on become parts of various die components. The primitive objects are recognised by their object numbers. Twelve object numbers are entered and stored.

3. The computer then asks for the type of process to be used. In this case, it is the compression process and so COM is typed in.

```
 R TGDESN

ENTER SCALE PARMTR:250

PROJ. PLANE : X-Y
ENTER PROD. REC. NO. :7871.

ENTER NO. OF.CONSTITUENT OBJS.:6

ENTER THEIR OBJ. NOS.,1 AT A TIME
**1

IF OBJ.    1  IS NEGATED,ENTER NEG. OBJ. NO.:7

**2

IF OBJ.    2  IS NEGATED,ENTER NEG. OBJ. NO.:8

**3

IF OBJ.    3  IS NEGATED,ENTER NEG. OBJ. NO.:9

**4

IF OBJ.    4  IS NEGATED,ENTER NEG. OBJ. NO.:10

**5

IF OBJ.    5  IS NEGATED,ENTER NEG. OBJ. NO.:11

**6

IF OBJ.    6  IS NEGATED,ENTER NEG. OBJ. NO.:12

WHAT PROCESS ? : COM

COMPR. M/C AVAI.: BIP2
**BIP2

BIP2 IS SUBP. 2
ENTER SHRINKAGE FACTOR (%) :0.9

PROD. SHAPE BECOMES DIE CAVITY (SUBP. 1) RP. AT 0.00      0.00      0.00
RPS. OF CONSTITUENT OBJS. ARE
      0.00       0.00       0.00
      0.00       0.00       0.00
    -42.38      -7.06       0.00
     42.38      -7.06       0.00
    -42.38      -6.05       0.00
     42.38      -6.05       0.00

HOW MANY CAVITIES ? : 1

IF ROT. RQRD ENTER AXIS NO. & ANGLE IN DEG.:

SUGGEST PONDER WELL DEPTH =30.0
NEW RPS. ARE
     0.000    -30.000      0.000
     0.000    -30.000      0.000
     0.000    -30.000      0.000
   -42.378    -37.063      0.000
```

```
     42.378    -37.063      0.000
    -42.378    -36.054      0.000
     42.378    -36.054      0.000

ENTER ANY CHANGE :

MENU FOR INSERTS : PIN,BOX
*BOX

ENTER CP AND BOX SIZES IN 6F10.0
0,-32,0,150,64,100

OBJ.     3      IS AN INSERT BOX AT   0.000000       -32.00000          0.00000
SIZES ARE       150.00000        64.00000       100.00000
REC. NO.=20101
IF ANY CHANGE R0RD,ENTER NEW CP & SIZES:


ENTER CLEARANCE ALLOWANCE : 0.01

OBJ.    4  REC. NO.= 20311  IS ITS NEG. EXPANSION (FOR CLEARANCE)
ENTER CP AND BOX SIZES IN 6F10.0
0,0,0,125,60,80

OBJ.     5      IS AN INSERT BOX AT       0.00000       0.00000
SIZES ARE       125.00000        60.00000        80.00000
REC. NO.=20521
IF ANY CHANGE R0RD,ENTER NEW CP & SIZES:


ENTER CLEARANCE ALLOWANCE : 0.01

OBJ.    6  REC. NO.= 20731  IS ITS NEG. EXPANSION (FOR CLEARANCE)
ENTER CP AND BOX SIZES IN 6F10.0


*PIN

PIN CODE : 1=TOP  2=BOT.
ENTER PIN CODE,SIZE,RP:
2,6,-42.378,-40.09,0

OBJ.    7  IS A PIN REC. NO.= 20941
SIZE=   6  RP AT     -42.38     -40.09       0.00
IF ANY CHANGE ENTER NEW CODE,SIZE,RP:


ENTER CLREARANCE ALLOWANCE : 0.01

OBJ.    8  REC. NO.= 21361  IS ITS NEG. EXPANSION (FOR CLEAR.)
ENTER PIN CODE,SIZE,RP:
2,6,42.378,-40.09,0

OBJ.    9  IS A PIN REC. NO.= 21781
SIZE=   6  RP AT      42.38     -40.09       0.00
IF ANY CHANGE ENTER NEW CODE,SIZE,RP:


ENTER CLREARANCE ALLOWANCE : 0.01

OBJ.   10  REC. NO.= 22201  IS ITS NEG. EXPANSION (FOR CLEAR.)
ENTER PIN CODE,SIZE,RP:


*
```

```
1<TOP> OR 2<BOTTOM> EJECTION ? : 1

THERE ARE 2 RETURN PINS, WHICH ARE
OBJ.   11   REC. NO. =   22621.   CP =   -152.000     12.500      0.000
OBJ.   12   REC. NO. =   23041.   CP =    152.000     12.500      0.000
ONLY ONE IS SHOWN
ALSO THERE ARE 2 PRESSURE PADS WHOSE REC. NO. ARE 13534 AND 13674
ENTER EJEC. PIN SIZE, RP:
8,-30,-37.063,0

OBJ.   13   IS AN EJEC. PIN REC. NO. = 23461.
SIZE=    8   RP AT     -30.00     -37.06       0.00
IF ANY CHANGE RQRD ENTER NEW SIZE, RP:


ENTER CLEARANCE ALLOWANCE : 0.01.

OBJ.   14   REC. NO. = 23881.   IS ITS NEG. EXPANSION <FOR CLEAR.>
ENTER EJEC. PIN SIZE, RP:
8,33\3\0,-37.063,0

OBJ.   15   IS AN EJEC. PIN REC. NO. = 24301.
SIZE=    8   RP AT      30.00     -37.06       0.00
IF ANY CHANGE RQRD ENTER NEW SIZE, RP:


ENTER CLEARANCE ALLOWANCE : 0.01. 0.00000          0.00000          0.00000

OBJ.   16   REC. NO. = 24721.   IS ITS NEG. EXPANSION <FOR CLEAR.>
ENTER EJEC. PIN SIZE, RP:


NEXT AVAILABLE REC. NO. = 25141.
STOP --

PAUSE --
TYPE <CR> TO EXIT


. CLOSE

.
```

Fig. 7.4.(1)

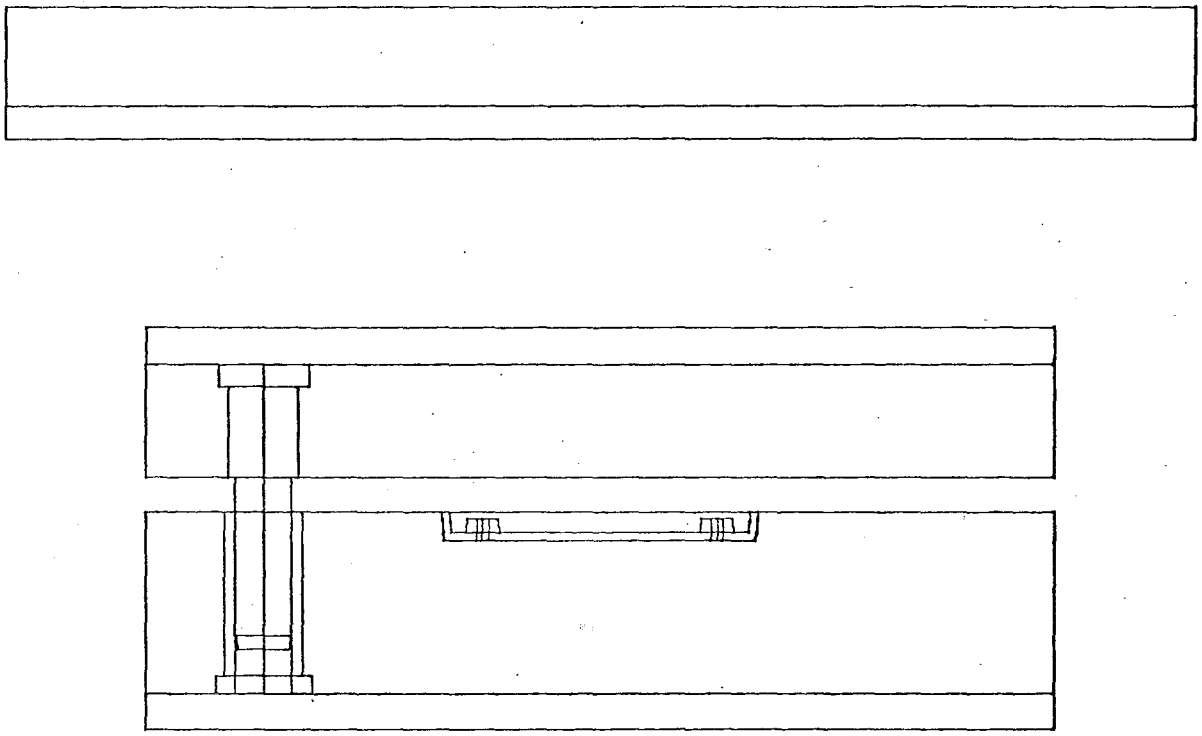


Fig. 7.4.(2)

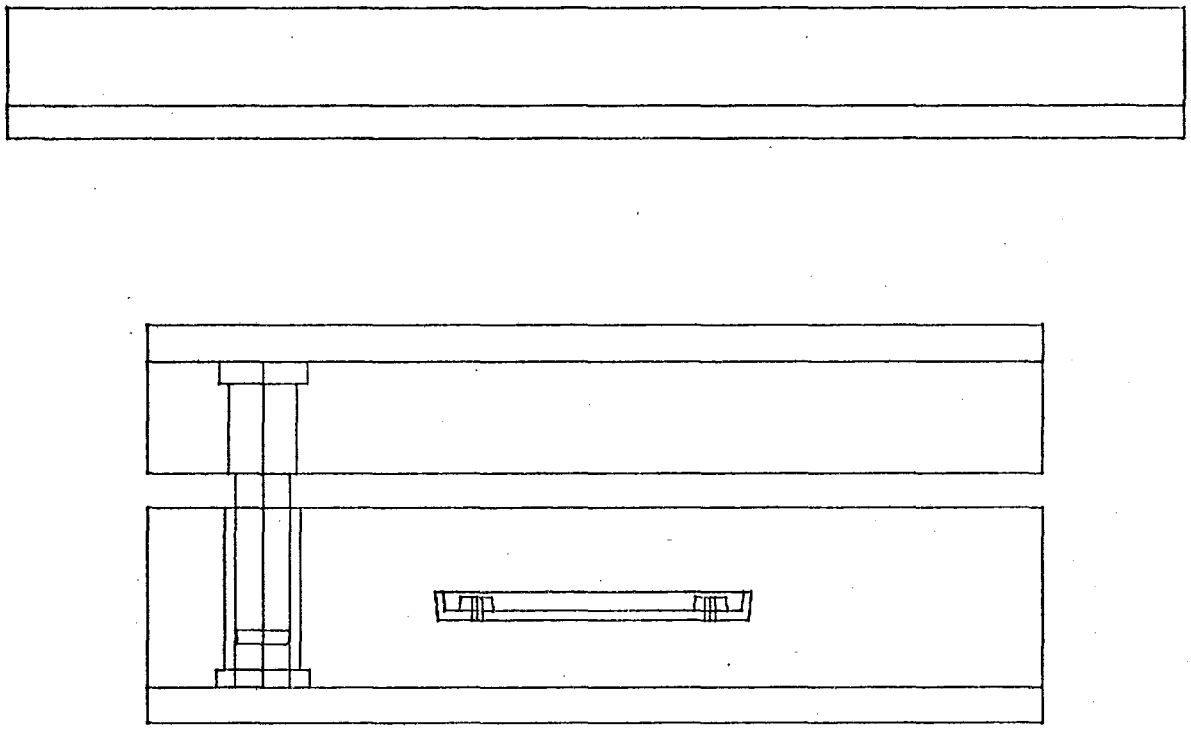


Fig. 7.4.(3)

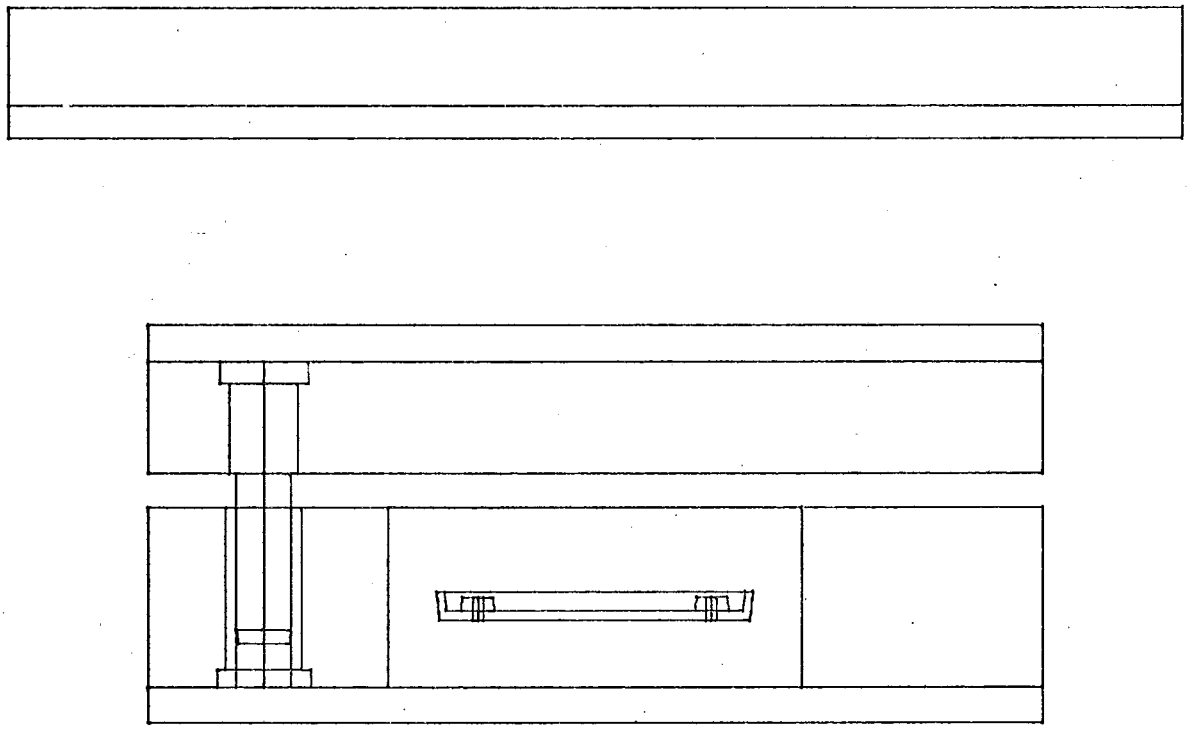


Fig. 7.4.(4)

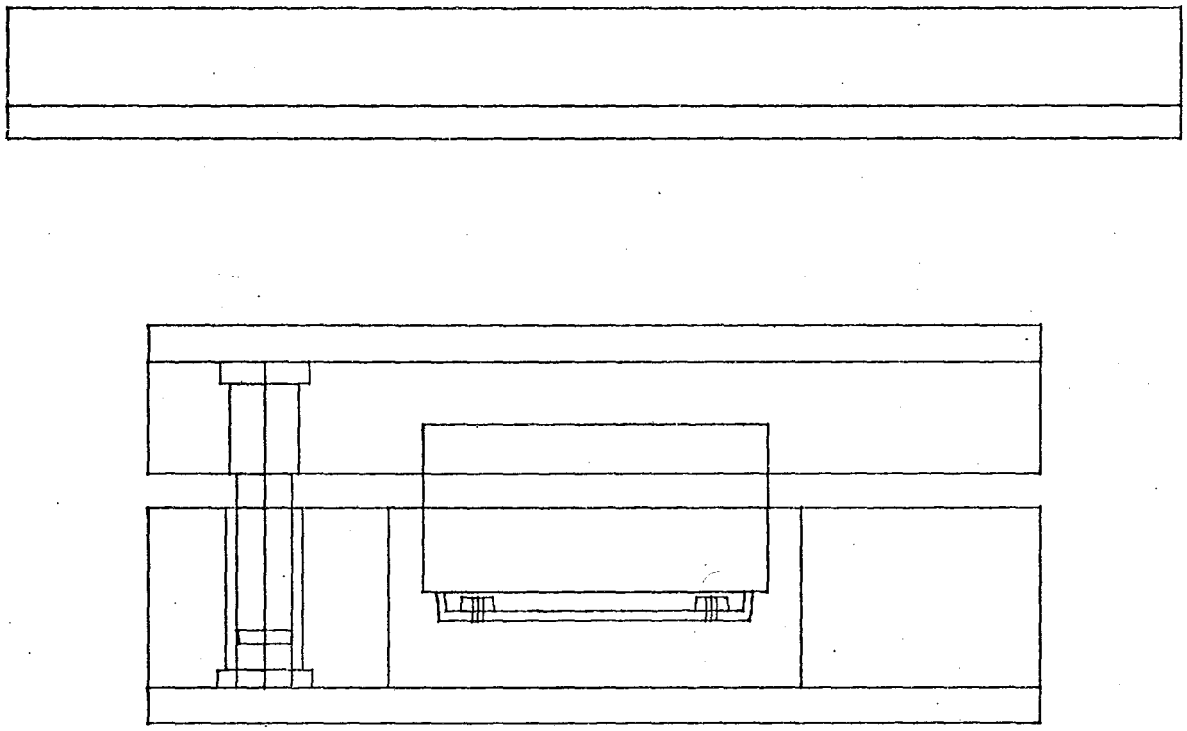


Fig. 7.4.(5)

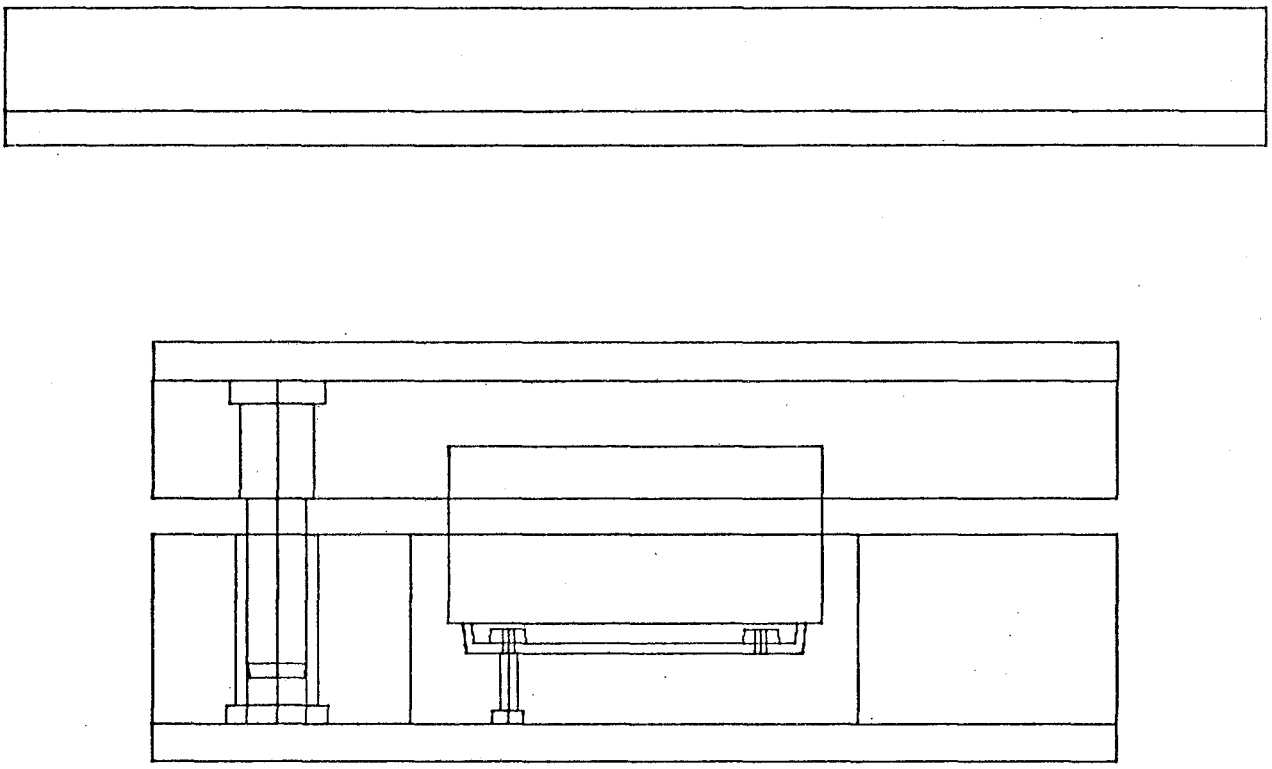


Fig. 7.4.(6)

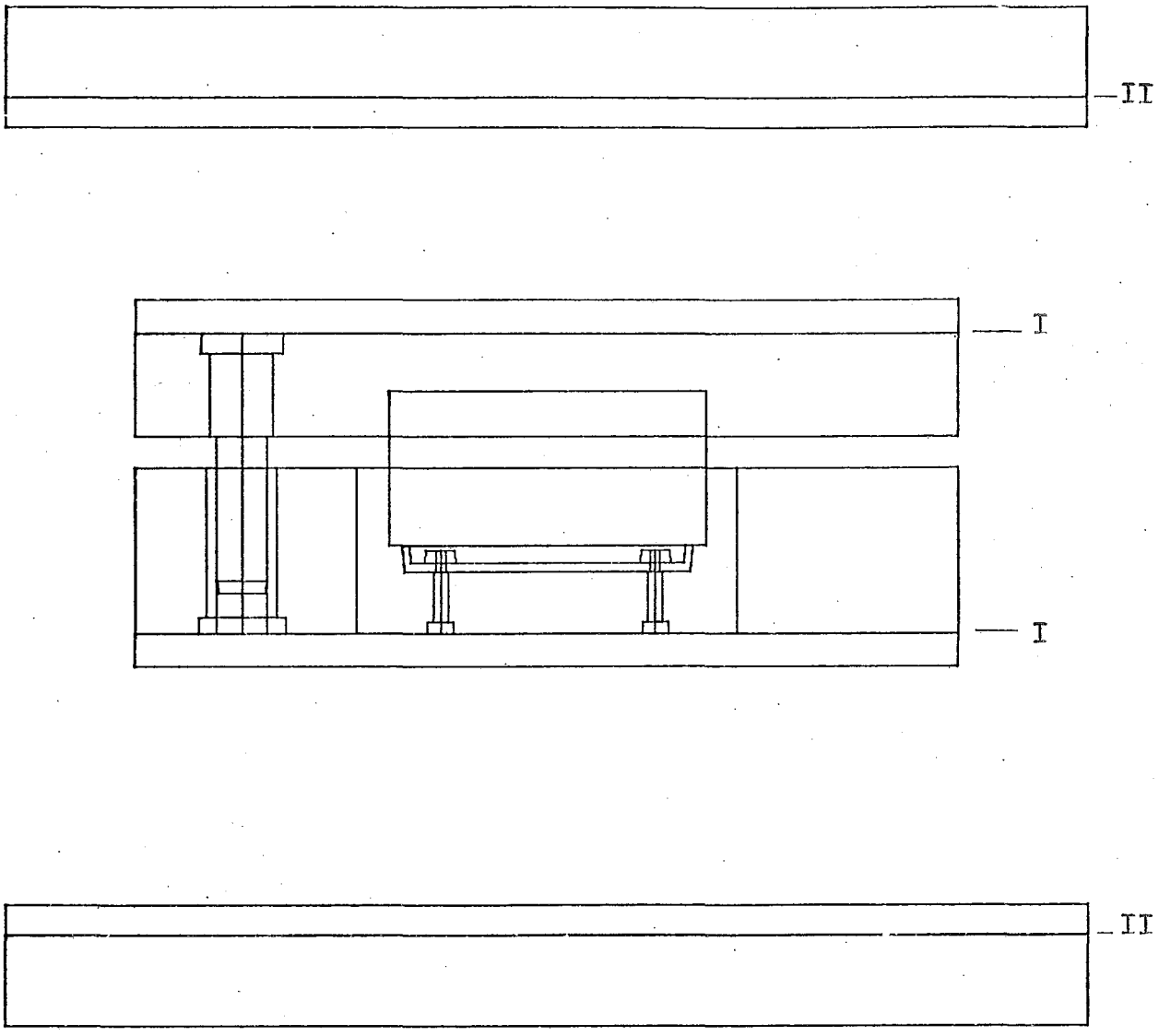


Fig. 7.4.(7)

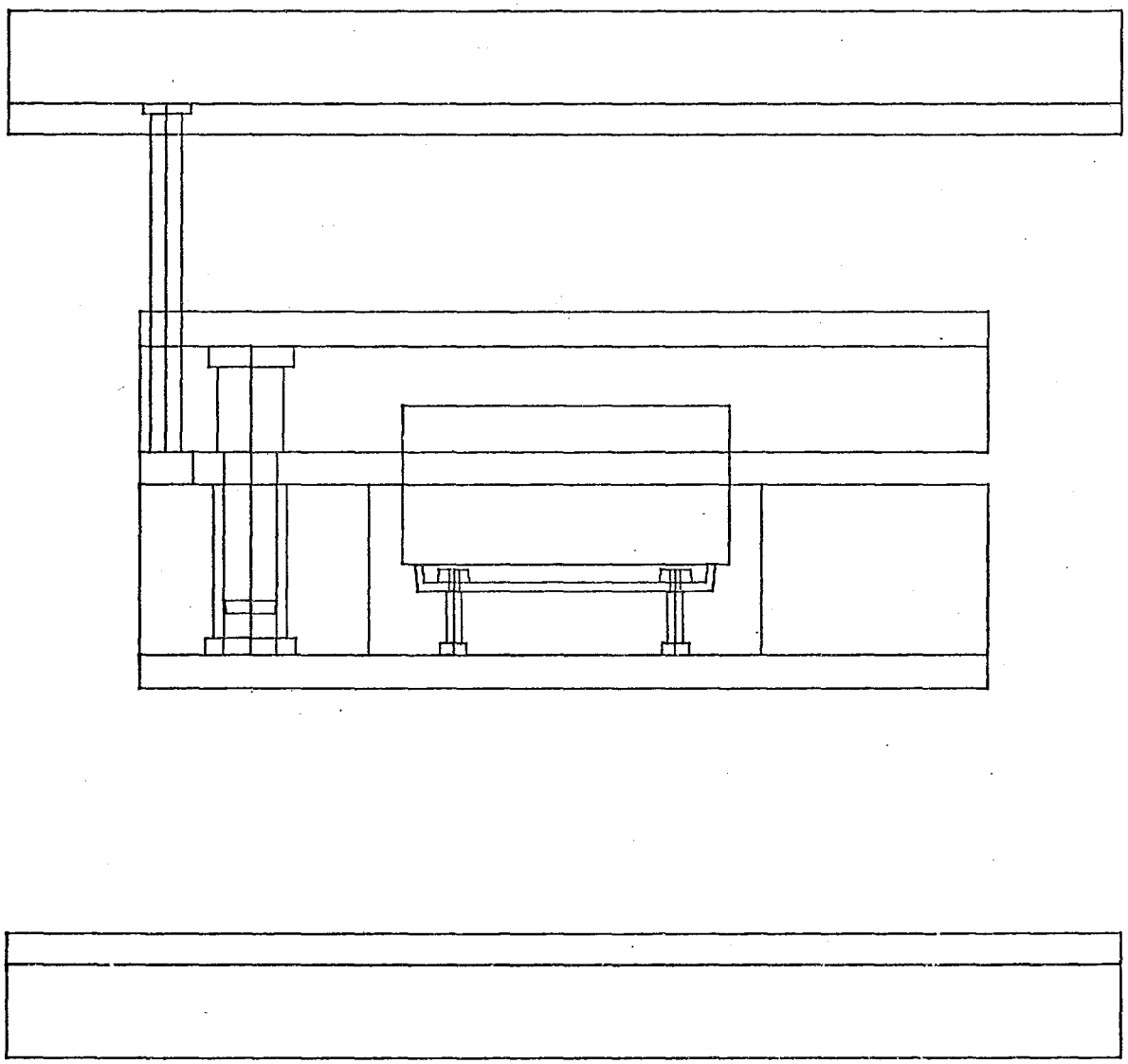


Fig. 7.4.(8)

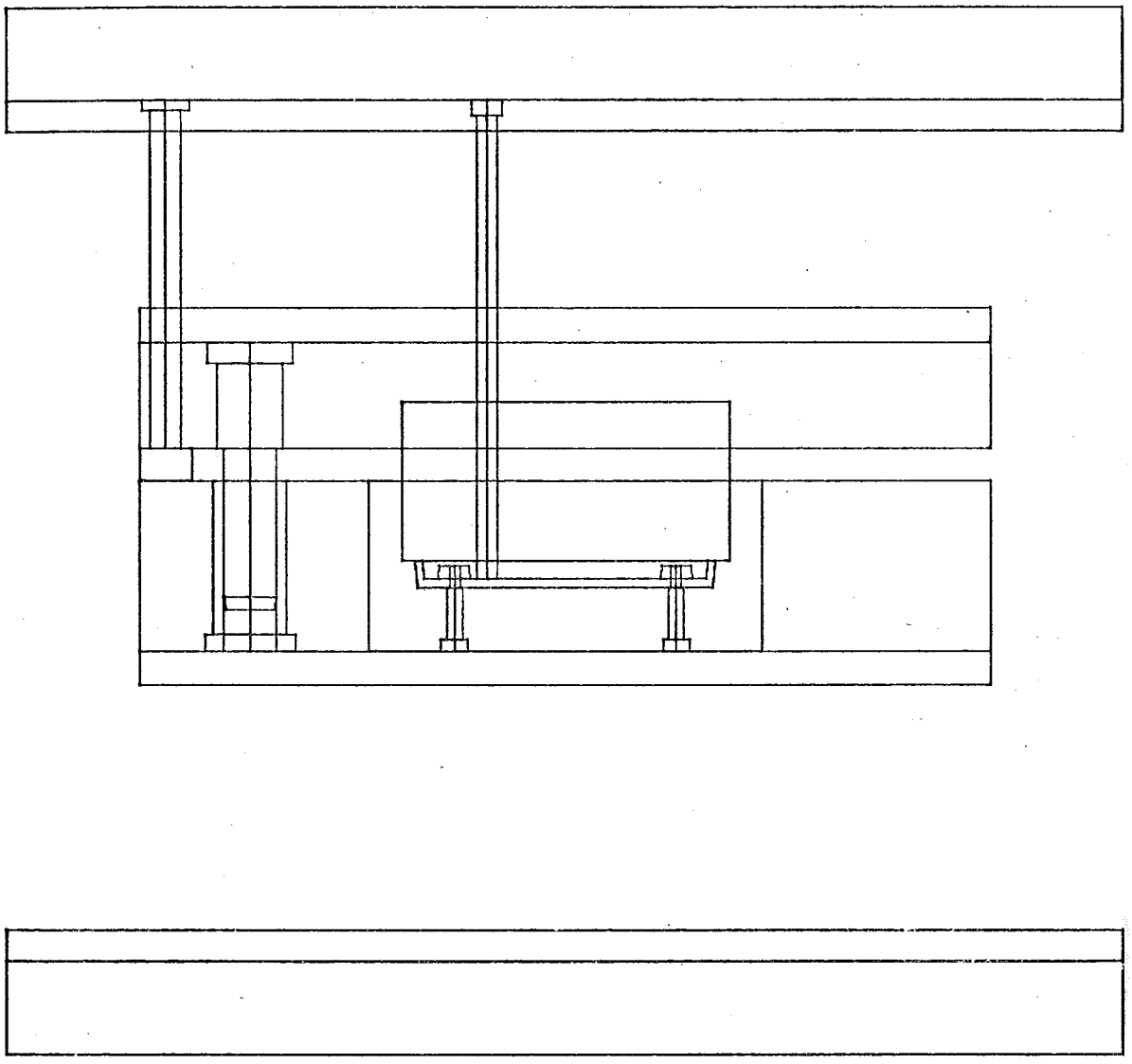


Fig. 7.4.(9)

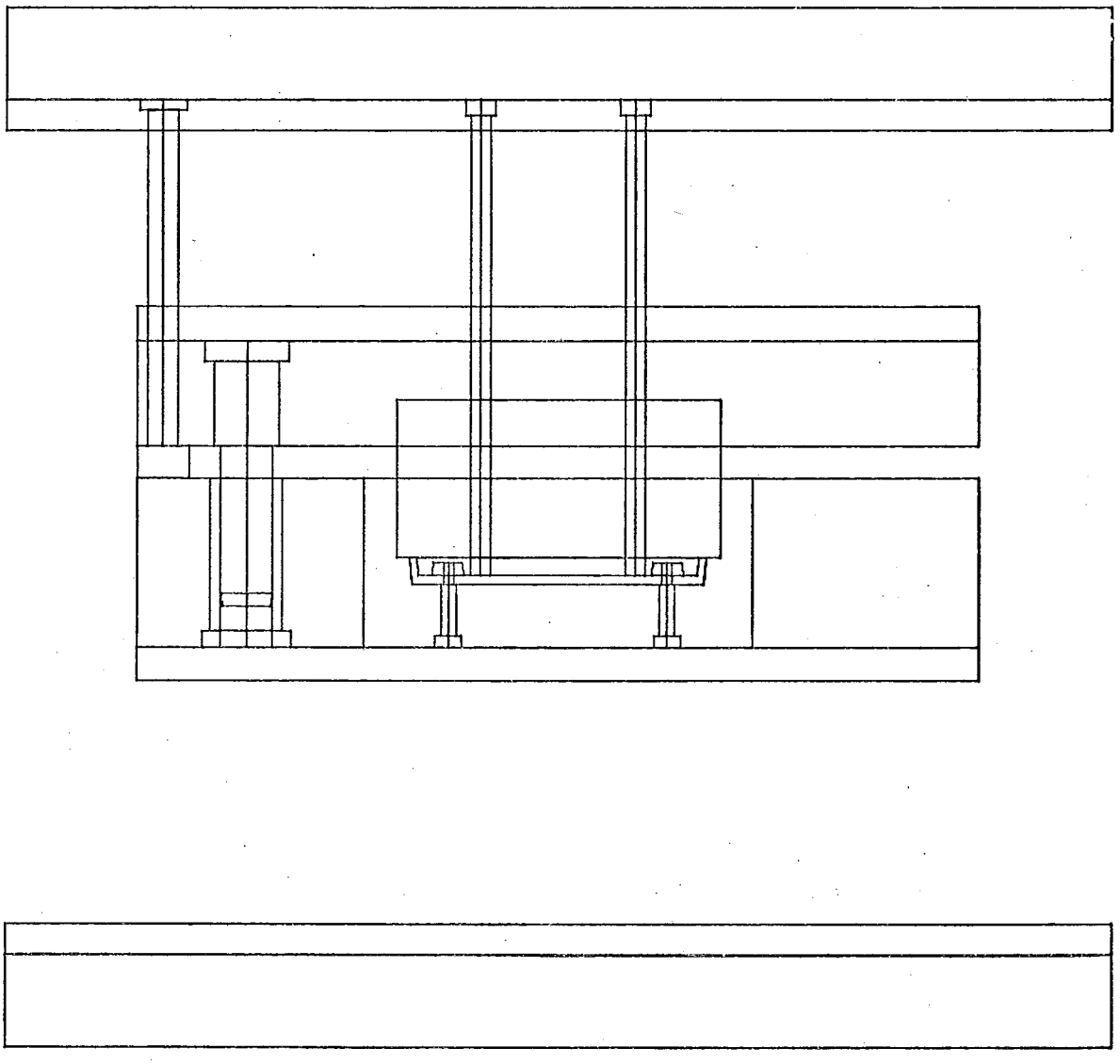


Fig. 7.4.(10)

4. The computer then lists all the compression machines which are available to the user and whose data has been previously stored on the disc. The user will take his choice. However, in this case, only the 20 ton Bipel machine is available and so BIP2 is typed in. The computer fetches the data structures of all components of the 20T Bipel machine and displays them on the screen (Fig. 7.4.(2)). A standard die set is also available for use with the Bipel and it comes as part of the machine data. The die set has four pairs of guide pins and guide bushes but only one pair is displayed. This is a common practice which the die designer often adopts in the conventional process. It is also unnecessary to display the screws which bolt the die blocks onto the machine platens. This saves room in the display buffer so that more important features can be displayed later on.

5. The computer now asks for the shrinkage factor of the product material. It could have looked up a table and obtained the shrinkage factor by itself if the product material had been entered in step 1. However, this would have cost more core storage and so the shrinkage factor is manually entered. In this case, for Urea Formaldehyde, it is 0.9%.

   It then expands every dimension on the product by 0.9%, including the primitive constituent objects. After the expansion, the product shape becomes the die cavity and the reference points of the cavity and the primitive objects are printed out.

6. Then the number of cavities is asked for. In this use, it is one.

7. The computer then asks if the user wishes to rotate the cavity in any other orientations. If he does, he enters the axis number of the rotation axis (e.g. 1 for OX axis, 2 for OY and 3 for OZ)

and the angle of rotation in degrees. In this case, the cavity is best left as it is and so just a <CR> key (Carriage Return) is typed and the process continues.

8. Powder Well Depth

The powder well depth could be calculated automatically if the bulk factor of the material and the volume of the cavity are known. The algorithm for calculating the volume of a general object was described by Braid. However, it is not implemented in this application. Instead, it is suggested that the powder well depth be 30.0 mm, which will put the cavity roughly half way inside the bottom die block. Again, the data structures of the cavity and its primitives are modified accordingly and the cavity is displayed at its new position in Fig. 7.4.(3).

The computer then waits for the user's decision by typing ENTER ANY CHANGE. If he wishes to change the powder well depth, he enters the new value here. In this application, no change is required and so a <CR> key is typed. The process continues.

9. Inserts

Now a menu for inserts is available. It consists of PIN and "BOX" inserts. Since the standard die set is made of ordinary steel, the cavity must be made from inserts of high quality tool steel. In this case a rectangular block is required and so "BOX" was typed in. The computer asks for and is given the defining parameters of the BOX. A BOX, object 3, is made (Fig. 7.4.(4)) enclosing the cavity and going through the bottom half of the standard die set. The computer then waits for any change to be made to this BOX insert. If no change is made, a <CR> is typed in. Since every insert is located in a slightly bigger hole in another component, the computer asks for the

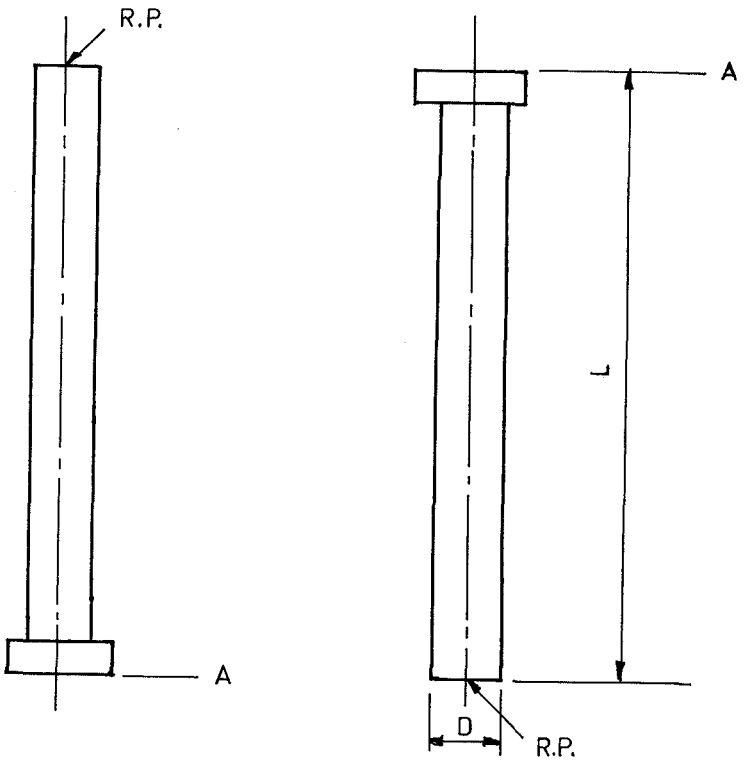
clearance allowance for this insert. Having got the clearance allowance, the insert is expanded along the OX and OZ directions and is made negative. The data structure of this negative shape (object 4) is stored separately. Object 4 will become a hole in the bottom half of the die set.

Similarly, another BOX insert with its negative object (object 6) is formed on the top half of the die set. The result is shown in Fig. 7.4.(5). When object 5 (the BOX insert) was made, the decision on the type of the mould had implicitly been made, i.e. the mould is semi-positive.

To make two cylindrical holes through the moulding, two insert pins are needed. They are object 7 and object 9. (Fig. 7.4.(6) and (7)). Objects 8 and 10 are their negative, expanded shapes respectively.

Insert pins (and ejector pins) are standard components whose data structures have been stored on the disc file. To define a pin, one would need to input the pin code, the diameter D, the length L and the R.P. (Fig. 7.5.). However, since face A of a pin would normally lie on a known plane the length L can be calculated automatically, being the distance from its R.P. to the known plane. All insert pins have face A lying on the machine platens (Face I, Fig. 4.7.(7)) while all ejector pins have face A lying on the ejector plate (face II, Fig.7.4.(7)). Therefore the user needs to enter only the pin code, the size (D) and the co-ordinates of the R.P.

After object 10 has been created, no more pins are required and so a <CR> is typed and the computer returns to the ready state by typing an asterisk (*). Also no more inserts are required, a <CR> is typed and the process continues.



(i) Pin Code = 2 (ii) Pin code = 1

Fig. 7.5 The Defining Parameters of a Standard Pin

10. Ejection Method

Continuing the process, the computer asks for a decision on the method of ejection. In this case, top ejection is required and so the code number 1 is typed in. The computer then fetches the data structure of two return pins and the two pressure pads. The return pins and the pressure pads are stored as part of the machine data. Again it is sufficient to display only one return pin and one pressure pad as shown in Fig. 7.4.(8). The computer then waits for the user to enter the defining parameters of the ejector pins. The parameters are the size (dia. D) and the co-ordinates of R.P. since the pin code has in fact been previously entered when the top ejection was decided. Two ejector pins (objects 13 and 15) together with their two negative, expanded shapes (objects 14 and 16) are created as shown in Fig. 7.4.(9) and (10).

At this stage, no more pins are required and a <CR> is typed.

The general die assembly is complete.

7.3 Extracting Individual Die Components

Having established the individual die assembly, the user can extract individual die components by adding and subtracting various objects through MERGING and INTERSECTION.

For example, to extract the top die block, the following steps are taken (Fig. 7.6.(a)):

- add positive PYR2 to object 5 by merging.
- subtract negative pins 14 and 16 from the resulting object by intersection.
- subtract negative CON3 and negative CON4 from the resulting object by merging.

The extracted top die block is shown in Fig. 7.6.(b). A computer print-out for the extraction of the top die block is shown below.

Similarly, the bottom die block is obtained by (Fig. 7.7.(a)):

- subtract the negative object 6 from the positive object 3 by intersection.
- subtract negative PYR1 from the resultant object by merging.
- subtract the negative pins 8 and 10 from the object by merging.

The extracted bottom die block is shown in Fig. 7.7.(b).

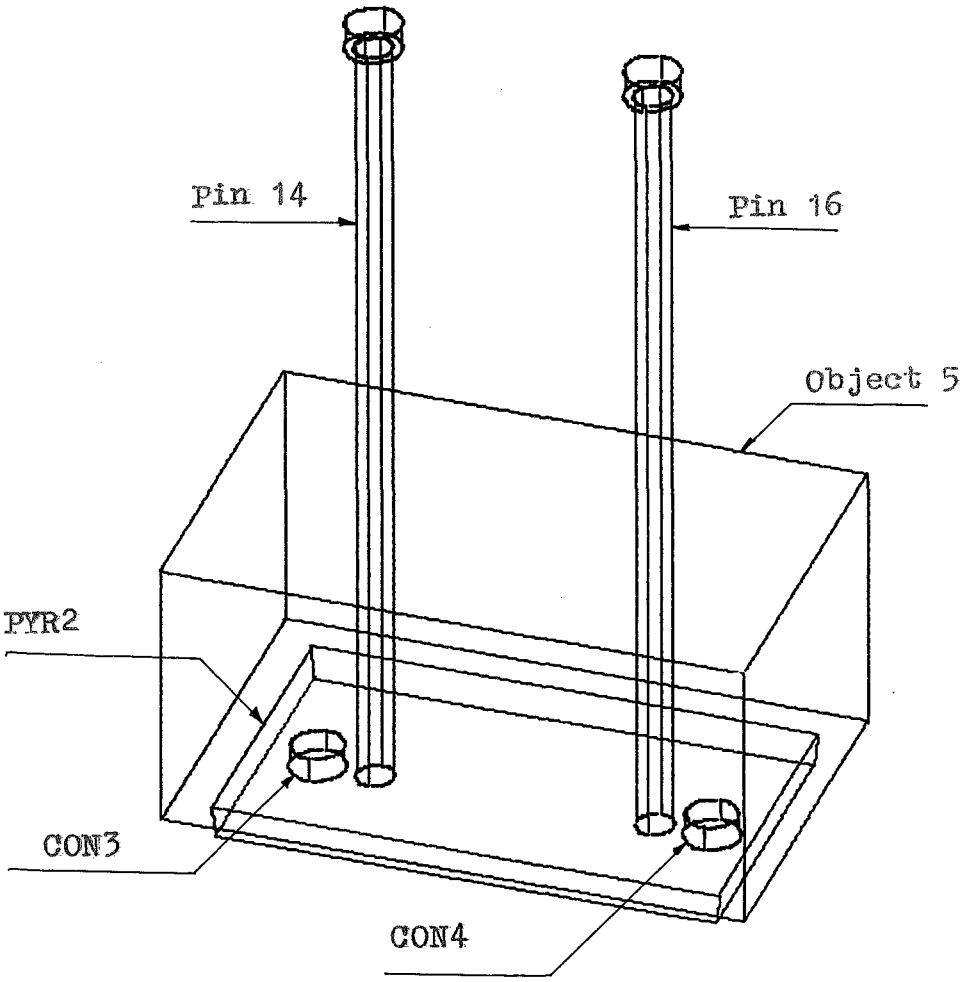
All other die components are extracted in a similar way.

Because there is a limited amount of the available core, a number of features are not shown in those drawings. For example, the cavity (bottom) die block shown in Fig. 7.8. should have a radius of 2 - 5 mm at the corner and a tapered angle θ between $\frac{1}{3}$ - 1^{o} [16]. Since these details are hard to see even if they were actually drawn, they are best specified on the drawing without having to draw them. This is again a common engineering practice. Another example is that although the

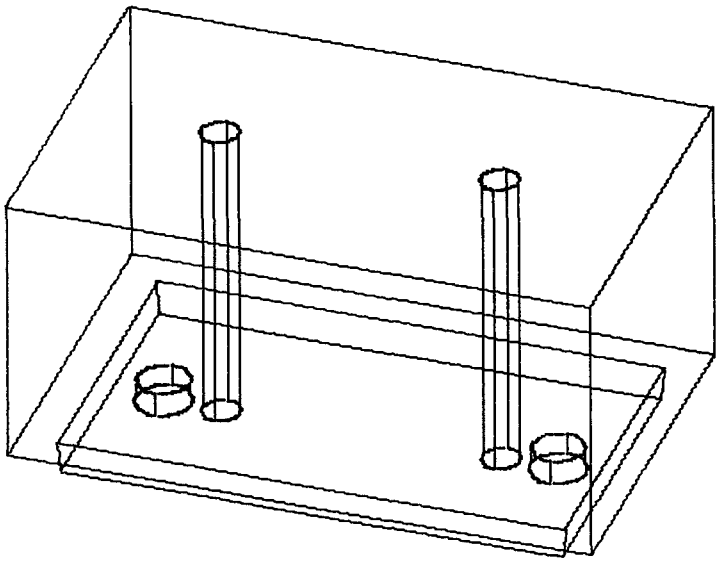
```
 ASS TT:LOG

.ASS TT:LST

.LOAD DA,TT

.R BATCH
*TGMERG/X


$JOB/RT11


ENTER REC. NOS. OF OBJ. 2 & OBJ. 1 :
?1661,20521

STOP --


STOP --


CURVED EDGE INTERSECTIONS ? :?

STOP --


STOP --


ENTER REC. NO. TO BE WRITTEN ON:?25141

STOP --


$EOJ

END BATCH

.R BATCH
*TGINTN/X


$JOB/RT11


ENTER OBJ. REC. NOS. OF OBJ. 2 & OBJ. 1 :
?23801,25141

STOP --


STOP --


STOP --
```

```
ENTER REC.  NO.  TO BE WRITTEN ON:?25141.

STOP --


$EOJ

END BATCH

.R BATCH
*TGINTN/X


$JOB/RT11


ENTER OBJ.  REC.  NOS.  OF OBJ.  2 & OBJ.  1 :
?24721,25141.

STOP --


STOP --


STOP ---


ENTER REC.  NO.  TO BE WRITTEN ON:?25141.

STOP --


$EOJ

END BATCH

.R BATCH
*1GMERG/X


$JOB/RT11


ENTER REC.  NOS.  OF OBJ.  2 & OBJ.  1 :
?31?1,25141.

STOP --


STOP ---


CURVED EDGE INTERSECTIONS ? :?

STOP ---
```

```
STOP --

ENTER REC.  NO.  TO BE WRITTEN ON:?25141.

STOP --


$EOJ

END BATCH

.R BATCH
*TGMERG/X


$JOB/RT11.

ENTER REC.  NOS.  OF OBJ.  2 & OBJ.  1 :
?3341,25141.

STOP --


STOP --


CURVED EDGE INTERSECTIONS ? :?

STOP --


STOP --


ENTER REC.  NO.  TO BE WRITTEN ON:?25143.

STOP --


$EOJ

END BATCH
```



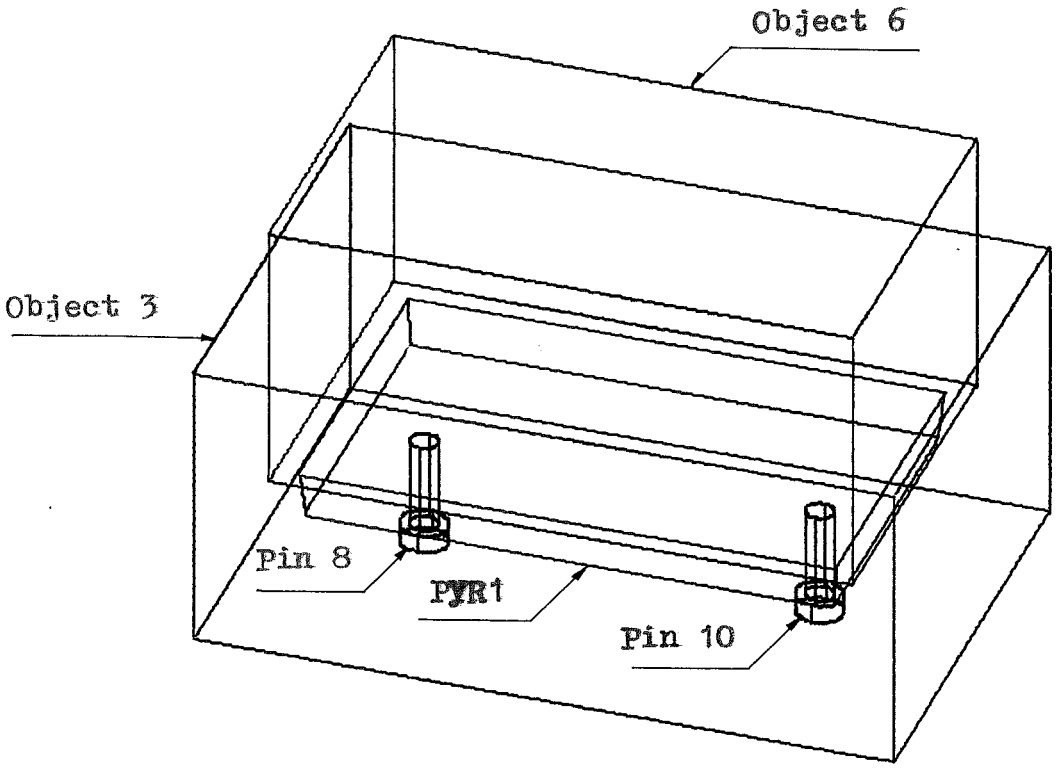
(a)



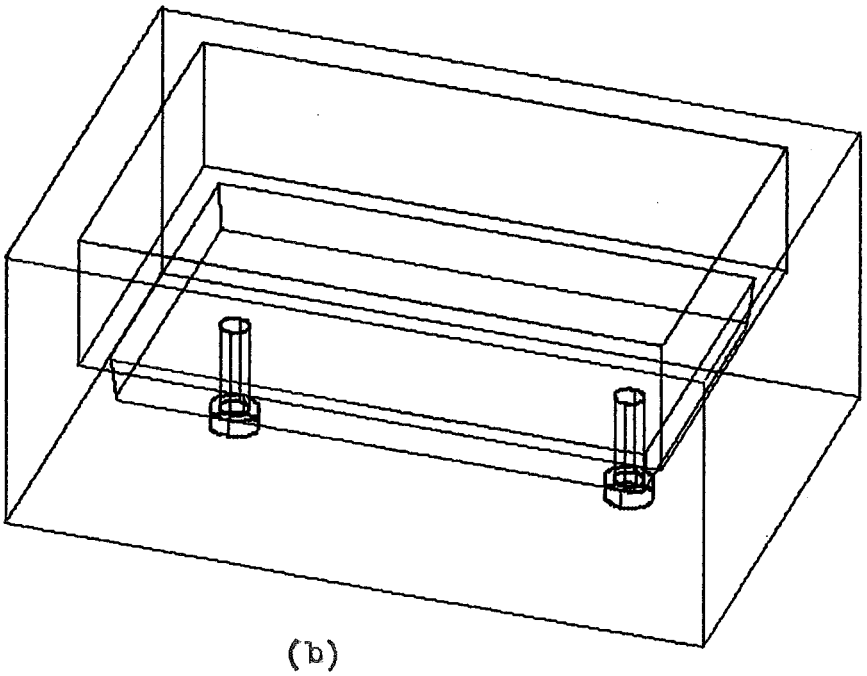
(b)

Fig. 7.6. Extracting The Top Die Block.

plunger (top die block) is bolted onto the top half of the die set, the screws and screw holes are not shown as they would have been if more core had been available.



(a)



(b)

Fig. 7.7. Extracting The Bottom Die Block.

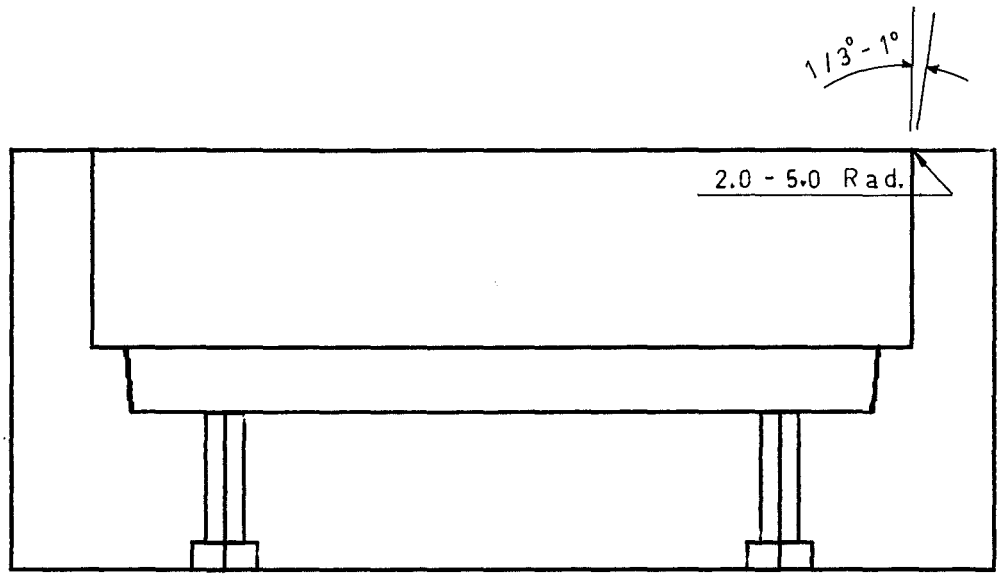


Fig. 7.8. An Orthogonal Projection of The Bottom Die Block.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

The programs demonstrate that the objectives of this research can be achieved. The design procedure was well laid out and could easily be followed by a designer skilful in the conventional way. Experience also shows that the method of describing objects in terms of simply-shaped volumes works particularly well in this application. The user would find this approach easy to use as it is very similar to the way a general die assembly is made. Even with a limited class of shapes, it has the power to describe a fairly complex object in a short time and it covers quite a range of shapes for plastic products.

The data structure employed in this application gives rapid processing of shapes bounded by flat, cylindrical and conical surfaces. The graphical display of the objects is quickly realised and the data is compact especially with the use of parametric, rational polynomial forms for cylindrical and conical surfaces and edges. However, there is one limitation of this method: a general curve of intersection between two curved surfaces cannot be represented.

The algorithms for adding and subtracting objects were further developed. They can add and subtract, very well, objects in this class of shapes. However, they were neither optimised nor rigorously studied to cover exceptionally complex objects.

The program versatility was severely restricted by the small size of the computer. A practical system ought to have a bigger memory to allow:

1. System continuity i.e. all programs can be combined into one single program.

2. A bigger range and a higher degree of complexity of the shapes. Objects should have a wider variety of surfaces.

3. A more flexible command language.

These and other areas that can be further developed are discussed below.

8.2 Future Work

If the restriction on the memory size is lifted, the programs can be easily combined to offer the user continuous operation from the start to finish. In linking the individual programs, a few more commands are needed: MER, INT, DIS, for example, to merge, intersect and display objects, respectively. Also data transfer between parts of the program would be more compact. For example, the objects can be recognised by their names and the user does not have to have a knowledge of their record numbers on disc.

Object, Definition and Modification

At the present, the position of an object is defined by the position of the reference point. This requires the user to have a knowledge of the positions of different reference points and have to keep track of them as they are modified. Very often, the user wishes to specify the positions by a command like "Make (or modify) the object B so that its reference point lies in the centre of face n of object A" or "........so that its reference point is at a (x, y, z) distance away from the reference point of object A" or "........ so that its reference point lines on the axis of object A with face m of object B touching face n of object A", etc. In other words, the user should have more freedom to specify a definition or a modification. It is easier for him to specify the relative position between features (e.g. reference point, faces, edges and vertices) of object with those of other objects. A command like this, however, requires that the

user knows the feature names (actually numbers) of objects.

This can simply be done by asking the computer to display them temporarily on the objects concerned before giving such a command, possibly with the use of a light pen.

Object Shapes

The class of shapes can be further widened to allow the description of a bigger range of plastic products. The shapes often have surfaces other than flat, cylindrical or conical surfaces. These surfaces can be defined in terms of perimeters (perimeter objects [1]), patches like Coons patches [2] or cross-sections [3, 8]. Care and attention should be paid to the use of these piece-wise representations regarding the degree of accuracy in the approximation they represent when they are modified. For example, when such a surface is scaled up, the number of pieces that approximate it should increase in order to retain the same degree of accuracy. However, the question of how to store those surfaces in the current data structure remains. Further study should concentrate on the use of a more comprehensive form of data structure bearing in mind that it has to continue dealing with common shapes efficiently.

Libraries and Disc Files

At the present, there is only one disc file serving as a back-up store containing all generated objects as well as libraries of machines and standard die parts. Because of the restriction on the maximum size of a disc file, it can be over-filled. To overcome this, more files can be created. For example, one separate file stores the available machines while another one stores standard die components, etc. It should be remembered that the opening of two many files can be expensive in terms of core usage and it is advisable, perhaps, to implement commands to close temporarily a currently-opened file before opening another one.

Automatic Extraction of Die Components

In the current system, the extraction of die components is done manually. The user has to know very well which objects penetrate which others in order to add or subtract them appropriately.

A possibility is left open to compile an automatic extraction algorithm or at least semi-automatic. The algorithm would have not only to sort out which objects penetrate or are in contact with other objects but also determine whether they should be added or subtracted. Perhaps certain rules of thumb, e.g. inserts must always be subtracted from die blocks, would help to simplify the algorithm a great deal.

Moving Pictures

The GT-44 graphics unit offers a further prospect for development of the system: the generation of moving pictures.

At the completion of the design, the complete moulding cycle can be watched in motion from the time plastics is injected or loaded until the moulding is ejected from the die cavity. At the same time, cam mechanisms and possibly engineering analyses such as compression pressures, clamping forces can be checked. The algorithm would involve a clear distinction between stationary components and the movable components. The degree of freedom for the movement of each movable component should also be recognised. The die platen opens or closes the cavity (in small discrete steps) and as it travels, it drives any movable components which come into contact and are in its way. This facility may bring a new feature in design: the ability to watch the die fully in operation even at the on-the-drawing-board stage.

Miscellaneous

The last areas that can be listed as future work are: the production of numerically-controlled machine tapes, dimensioned drawings and the computation of engineering quantities.

The internal data structure of an object can be used with additional machining information to produce NC machine tapes for automatic production of die components. To do this, a hole would have to be recognised as a separate entity rather than just a space bounded by some negative faces.

Even when the components can be automatically produced by NC tapes, dimensioned drawings are still useful for assembly and checking. It would still be a big improvement if dimensional drawings can be automatically produced. The facility should also allow a user to cross-hatch a cut section and edit dimension lines, positions, again possibly with a light pen.

Engineering quantities, such as volumes, areas, centres of pressure, etc. are required to calculate the powder well depth, the clamping force, etc. in a die design. These calculation routines should be able to accept the current data structure as input. Braid [1] did actually give an algorithm for calculating the volume of an object having the current form of data structure.

APPENDIX A

THE DESCRIPTIONS OF UNIT OBJECTS

There are six unit objects: a box, a cylinder, a wedge, a fillet, a truncated cone and a truncated pyramid as shown in Fig. 3.1.

I. BOX:

Name: BOX

The name of a unit object is a three letter word recognised by the computer program which will fetch the corresponding routine on receiving it.

Fig. A.1 shows all details of the vertices, the edges and the surfaces of the box. The unit box has its reference point being its centroid at the origin and it is a cubic box of 2-unit long edges. A detailed description of the data structure is given in Appendix B.

On calling the routine, the user is required to enter three dimensions of the edges along the x, y, z axes and the co-ordinates of the reference point. Since for this application, it is almost always required that the axes of symmetry of the box be parallel to the co-ordinate axes and hence the rotation procedure is not in this routine. The mechanism of the routine then consists of setting a matrix S, the scaling matrix, a matrix [TL], the translational matrix and finally the transformation matrix [TR] = S[TL].

With [TR], all data of the unit box is transformed to give a data description of the required primitive box.

II. CYLINDER

Name: CYL

The construction of the unit cylinder is shown in Fig. A.2. The unit cylinder has its axis as the z axis. It has a diameter of two units long and a length of two units long. Its reference point, being its centroid, is at the origin.

On calling the routine, the user is required to enter the reference point, the length, the diameter and the axis vector of the primitive object. The routine will then generate a primitive cylinder according to these specifications.

III. WEDGE

Name: WED

Fig. A.3 shows the detailed construction of the unit edge. It has edge 7 lying on the y-axis. The mid-point of edge 7 is its reference point, being the origin. The dimensions along the x, y, z axes are all 1.0 unit long.

Again in this application, it is only required that the wedge should have its edge 7 parallel to one of the three co-ordinate axes.

To define a primitive wedge, the following parameters are required:

1. The axis number IA: It can be any one of the numbers 1, 2, 3 to specify that the wedge will have its edge 7 parallel to the x, y, or z axes respectively.
2. The quadrant number IQ: It can be any one of the numbers 1, 2, 3, 4 to specify whether the wedge is in quadrant 1, 2, 3, or 4 respectively when looking down against the direction of the axis IA. e.g. the unit wedge could have IA = 2, IQ = 1.
3. The three dimensions of the wedge measured along the x, y, z axis respectively.
4. The co-ordinates of the reference point.

IV. FILLET

Name: FIL

The construction of the unit fillet is shown in Fig. A.4. It is similar to the unit wedge except that face 5 is now a cylindrical face whose radius is 1.0 unit long.

To define a primitive fillet the following parameters are required:

1. The axis number IA: the same as for the wedge.

2. The quadrant number IQ: the same as for the wedge.

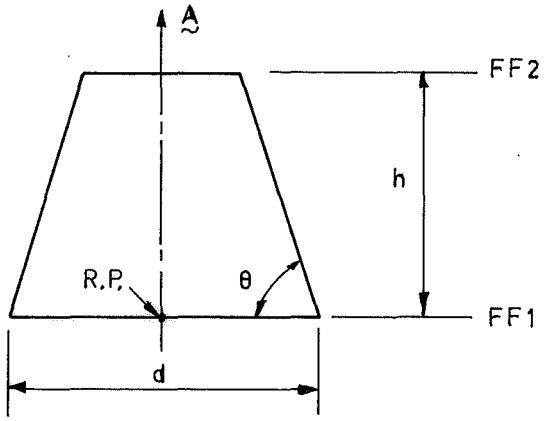
3. The two dimensions which are the height of the wedge and the radius of the cylindrical surface.

4. The co-ordinates of the reference point.

V. THE TRUNCATED CONE:

Name: CON

Fig. A.5 shows the detailed construction of the unit truncated cone. It has a base diameter of 2 unit lengths, reference point being the centre of the base circle. The base is the flat face 1, marked FF1 in the figure. The primitive CON is defined by the parameters shown in the figure below.



d: base diameter

θ: base angle

h: height

RP: reference point being the centre of the base circle.

Parameters defining a primitive CON.

Since θ is a variable, vertices 5, 6, 7, 8 lying on flat face 2 will vary and therefore are not stored in the unit CON. For the same reason, the unit CON does not have a fixed height h and does not have the equation of FF2, nor the L numbers in group 2.

When using CON, the user is required to enter:

1. The co-ordinates of the RP.
2. The base diameter d.
3. The height h.
4. The base angle θ.
5. The components of the axis vector $\underset{\sim}{A}$.

Routine CON, on receiving these parameters, will calculate and fill in all the missing data.

VI. TRUNCATED PYRAMID

Name: PYR

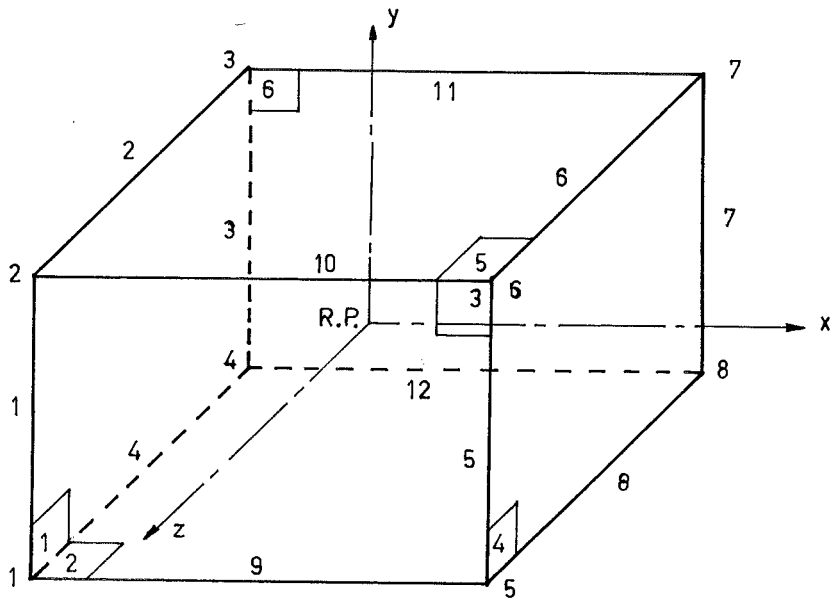
Fig. A.6 shows the construction of the unit truncated pyramid, which is exactly like the unit box except the co-ordinates of the vertices and the equations of the surfaces. Fig. A.6 also shows the parameters defining a truncated pyramid.

1. Three dimensions AX, AY, AZ.
2. Two tapered angles θ_1, θ_2.
3. Co-ordinates of the RP.
4. The components of the axis vector $\underset{\sim}{I}$.

Because of the variability of θ_1 and θ_2, the co-ordinates of the vertices and the equation of the surfaces are not stored, but will be generated by PYR at the time it receives the above parameters. The rest of the data structure is exactly like that of the box. Therefore, when PYR is called, it reads the stored data of the unit box, but it ignores the co-ordinates of the vertices and the equations of the surfaces and it generates the correct ones by itself.

Although the user can alternatively create a truncated pyramid by merging one positive box and four negative wedges, it is much faster and easier to use PYR. Besides the shape of a truncated pyramid is so frequently used in die assembly and plastic products that it is worth making a separate routine just to generate this particular shape.

The reason why tapered angles are used to specify a CON and a PYR is because they are usually small (generally in the range 5^{o} - 10^{o}) in this application. It would be awkward to specify such a CON or a PYR in terms of length dimensions.



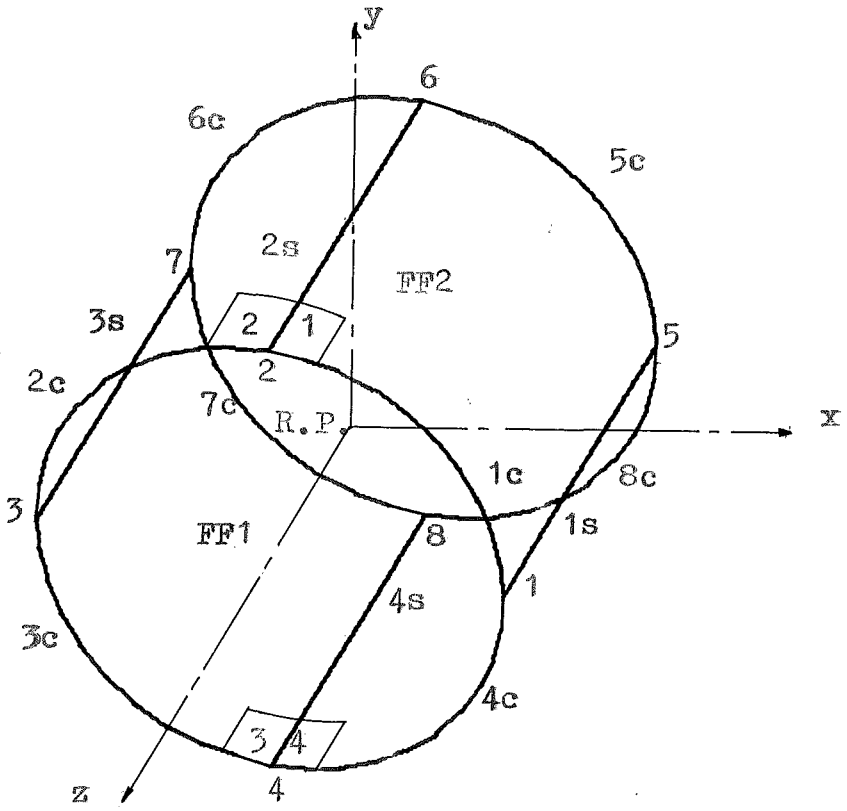
Vertex	X	Y	Z
1	-1.	-1.	1.
2	-1.	1.	1.
3	-1.	1.	-1.
4	-1.	-1.	-1.
5	1.	-1.	1.
6	1.	1.	1.
7	1.	1.	-1.
8	1.	-1.	-1.

	Face Equation			
Face	A	B	C	D
1	-1.	0.	0.	-1.
2	0.	-1.	0.	-1.
3	0.	0.	1.	-1.
4	1.	0.	0.	-1.
5	0.	1.	0.	-1.
6	0.	0.	-1.	-1.

Face	Edges			
1	1	2	3	4
2	4	12	8	9
3	1	9	5	10
4	8	7	6	5
5	2	10	6	11
6	3	11	7	12

Edge	Vertex 1	Vertex 2
1	1	2
2	2	3
3	3	4
4	4	1
5	6	5
6	7	6
7	8	7
8	5	8
9	5	1
10	6	2
11	7	3
12	8	4

Fig. A.1 The Construction of the Unit Box



Vertex	1	2	3	4	5	6	7	8
x	1.	0.	-1.	0.	1.	0.	-1.	0.
y	0.	1.	0.	-1.	0.	1.	0.	-1.
z	1.	1.	1.	1.	-1.	-1.	-1.	-1.

Straight Edge	1S	2S	3S	4S
Vertex 1	1	6	7	8
Vertex 2	5	2	3	4

	Edges having L no. group 1				Edges having L no. group 2			
Curved edge	1C	2C	3C	4C	5C	6C	7C	8C
Vertex 1	1	2	3	4	6	7	8	5
Vertex 2	2	3	4	1	5	6	7	8

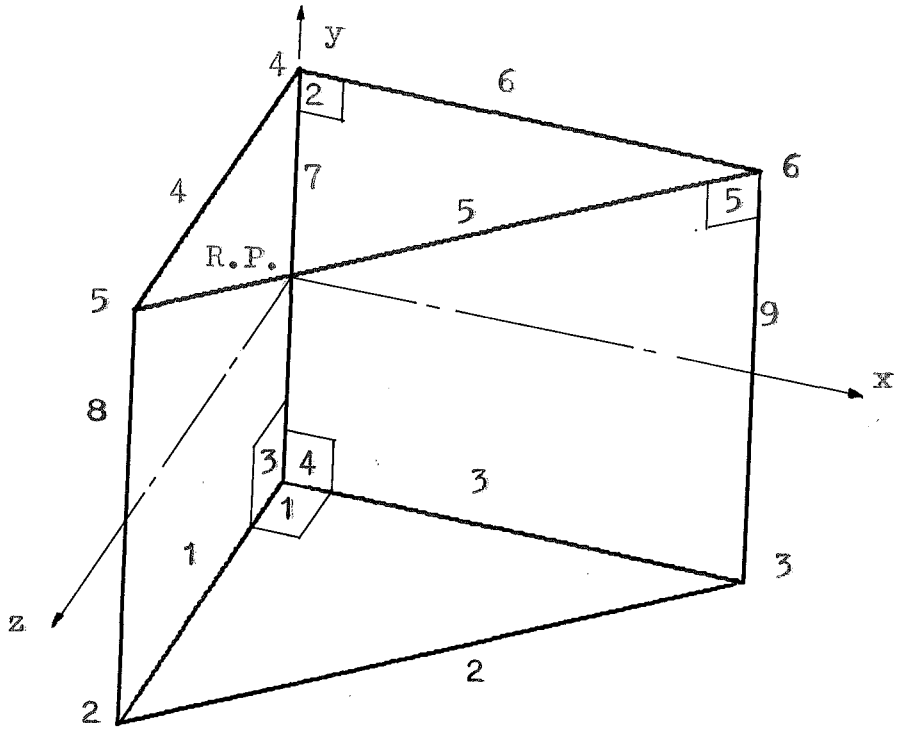
Face		Edges			
1	(1 Cyl.)	1C	1S	5C	2S
2	(2 Cyl.)	2C	2S	6C	3S
3	(3 Cyl.)	3C	3$	7C	4S
4	(4 Cyl.)	4C	4S	8C	1S
5	(1 Flat)	1C	2C	3C	4C
6	(2 Flat)	5C	8C	7C	6C

Flat Face	Face Equation A	B	C	D
1	0.	0.	1.	-1.
2	0.	0.	-1.	-1.

Cylindrical Face	Parametric Equation (rounded-off)		
1	-0.41421	-0.41421	0.
	-0.58579	1.41421	0.
	1.	0.	0.
	0.	0.	-2.
	0.	0.	1.
2	0.41421	-0.41421	0.
	-1.41421	-0.58579	0.
	0.	1.	0.
	0.	0.	-2.
	0.	0.	1.
3	0.41421	0.41421	0.
	0.58579	-1.41421	0.
	-1.	0.	0.
	0.	0.	-2.
	0.	0.	1.
4	-0.41421	0.41421	0.
	1.41421	0.58579	0.
	0.	-1.	0.
	0.	0.	-2.
	0.	0.	1.

Group	L-number			
1	0.	0.	0.	0.
2	0.	0.	0.	1.

Fig. A.2 The Construction of the Unit Cylinder



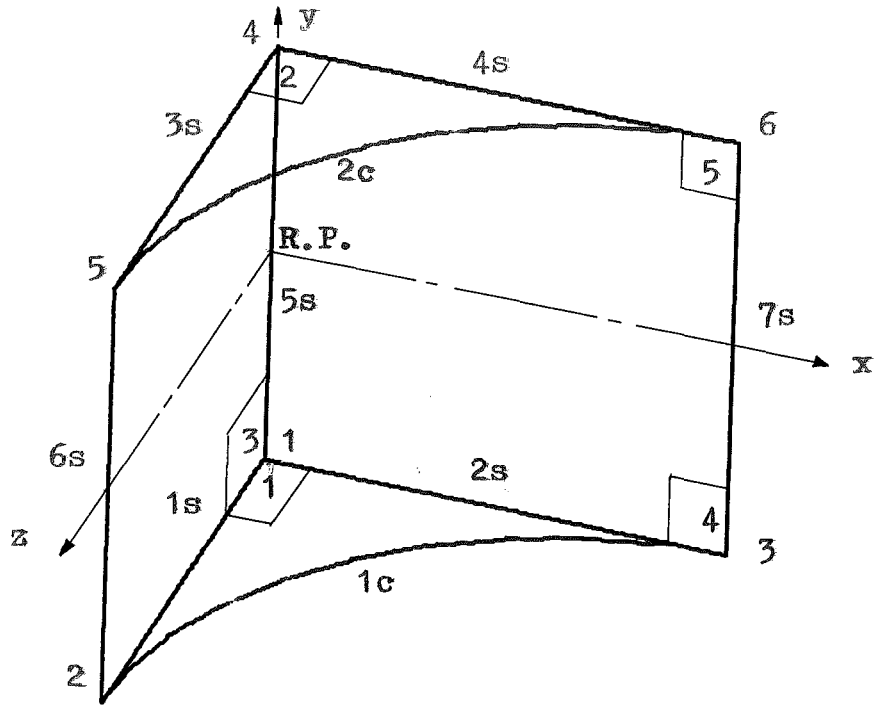
Vertex	1	2	3	4	5	6
x	0.	0.	1.	0.	0.	1.
y	-0.5	-0.5	-0.5	0.5	0.5	0.5
z	0.	1.	0.	0.	1.	0.

Edges	1	2	3	4	5	6	7	8	9
Vertex 1	2	3	1	4	5	6	4	2	6
Vertex 2	1	2	3	5	6	4	1	5	3

Face	Edges			
1	1	3	2	
2	4	5	6	
3	1	8	4	7
4	6	9	3	7
5	8	2	9	5

	Face Equation			
Face	A	B	C	D
1	0.	-1.	0.	-0.5
2	0.	1.	0.	-0.5
3	-1.	0.	0.	0.
4	0.	0.	-1.	0.
5	1.	0.	1.	-1.

Fig. A.3. The Construction of the Unit Wedge



Vertex	1	2	3	4	5	6
x	0.	0.	1.	0.	0.	1.
y	-0.5	-0.5	-0.5	0.5	0.5	0.5
z	0.	1.	0.	0.	1.	0.

Straight Edge	1S	2S	3S	4S	5S	6S	7S
Vertex 1	2	1	4	6	4	2	6
Vertex 2	1	3	5	4	1	5	3

CurvedEdge	1C	2C
Vertex 1	3	5
Vertex 2	2	6
L Group No.	1	2

Face	Edges
1 (1 Flat)	1S 2S 1C
2 (2 Flat)	3S 2C 4S
3 (3 Flat)	1S 6S 3S 5S
4 (4 Flat)	4S 7S 2S 5S
5 (1 Cyl.)	6S 1C 7S 2C

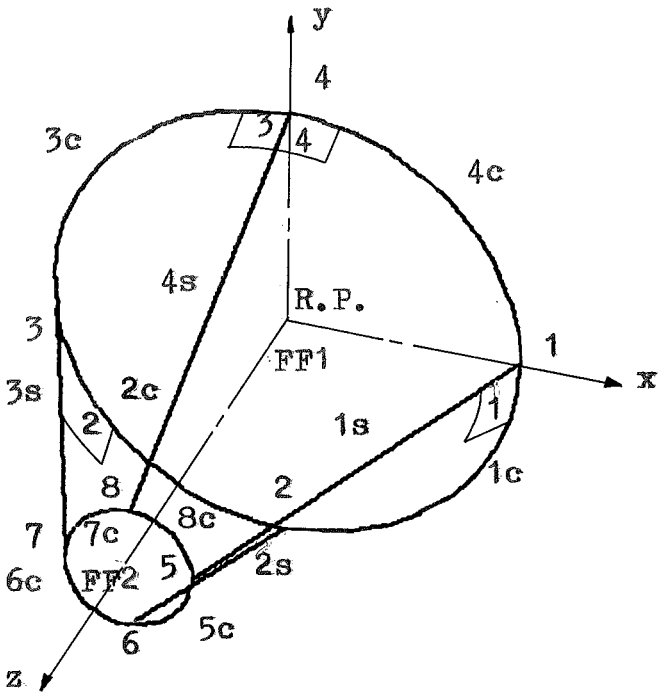
Flat Face	Face Equation A	B	C	D
1	0.	-1.	0.	-0.5
2	0.	1.	0.	-0.5
3	-1.	0.	0.	0.
4	0.	0.	-1.	0.

Parametric Equation of CurvedFace 1:

0.41421	0.	0.41421
0.58579	0.	-1.41421
-1.	0.	0.
0.	1.	0.
1.	-0.5	1.

Group	L-Numbers			
1	0.	0.	0.	0.
2	0.	0.	0.	1.

Fig. A.4 The Construction of the Unit Fillet



Vertex	1	2	3	4
x	1.	0	-1.	0.
y	0.	-1.	0.	1.
z	-1.	-1.	-1.	-1.

Straight Edge	1S	2S	3S	4S
Vertex 1	1	6	7	8
Vertex 2	5	2	3	4

Curved Edge	1C	2C	3C	4C	5C	6C	7C	8C
Vertex 1	1	2	3	4	6	7	8	5
Vertex 2	2	3	4	1	5	6	7	8
L group no.	1	1	1	1	2	2	2	2

Face	Edges			
1 (1 Con.)	1C	1S	5C	2S
2 (2 Con.)	2C	2S	6C	3S
3 (3 Con.)	3C	3S	7C	4S
4 (4 Con.)	4C	4S	8C	1S
5 (1 flat)	1C	2C	3C	4C
6 (2 flat)	5C	8C	7C	6C

Flat Face	Face Equation			
	A	B	C	D
1	0.	0.	-1.	-1.
2	0.	0.	0.	0.

Conic Face	Parametric Equation		
1	-0.41421	0.41421	0.
	-0.58579	-1.41421	0.
	1.	0.	0.
	0.	0.	2.
	0.	0.	-1.
2	0.41421	0.41421	0.
	-1.41421	0.58579	0.
	0.	-1.	0.
	0.	0.	2.
	0.	0.	-1.
3	0.41421	-0.41421	0.
	0.58579	1.41421	0.
	-1.	0.	0.
	0.	0.	2.
	0.	0.	-1.
4	-0.41421	-0.41421	0.
	1.41421	-0.58579	0.
	0.	1.	0.
	0.	0.	2.
	0.	0.	-1.

Group	L-Numbers					
1	0.	0.	0.	-1.17157	1.17157	-2.0
2	0.	0.	0.	0.	0.	0.

Fig. A.5 The Construction of the unit truncated cone

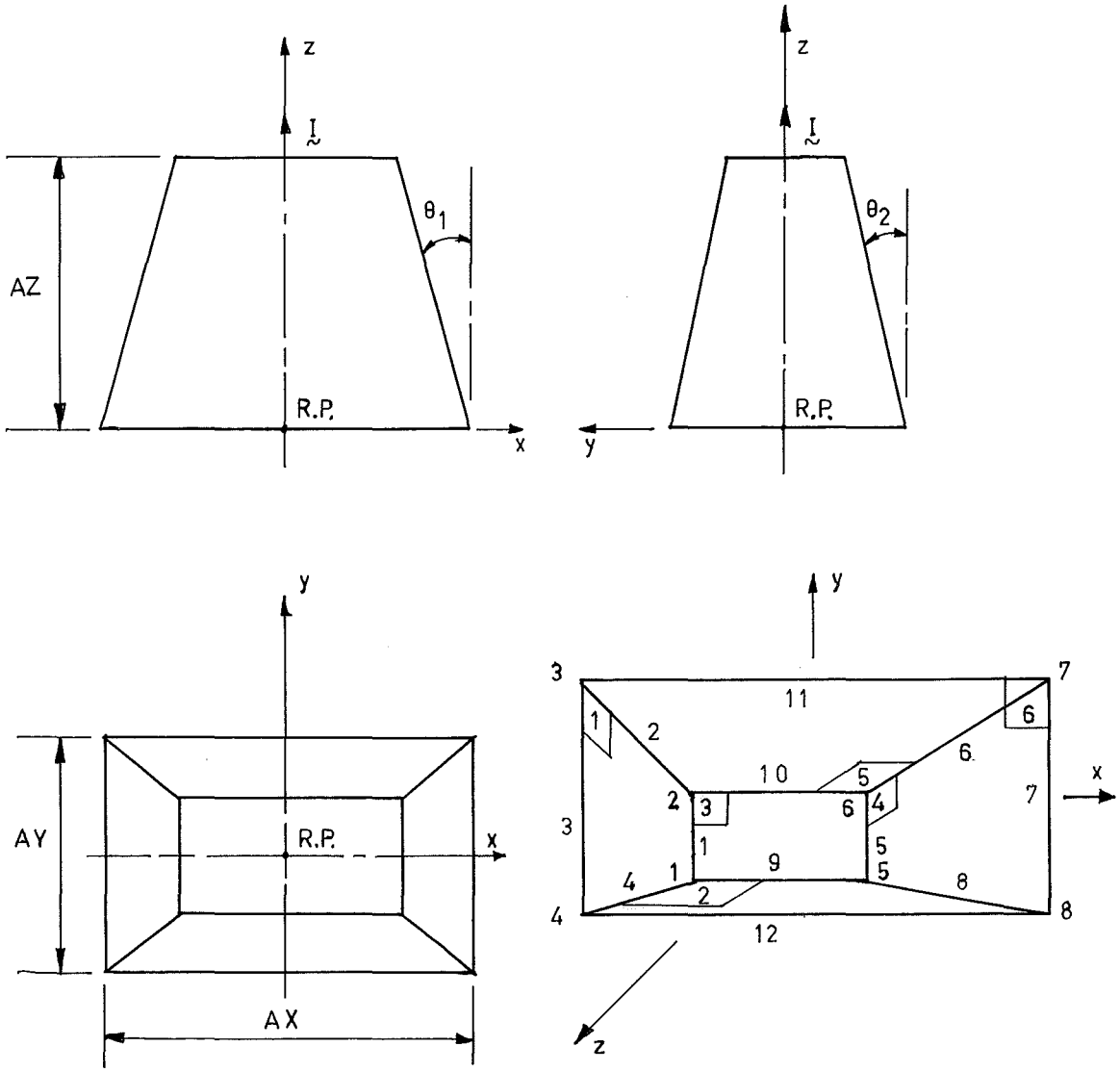


Fig. A-6 The Construction of The Unit Truncated Pyramid

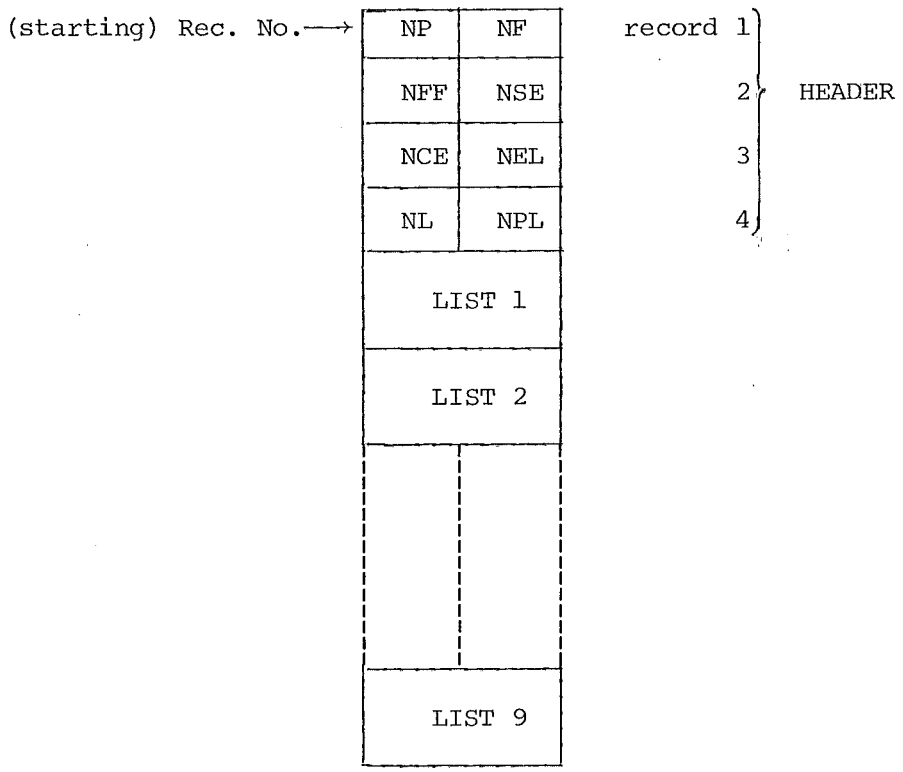
APPENDIX B

DATA STRUCTURE

The data structure representing one object is held in a data block on the disc. Each data block begins with a 4-record header, followed by 9 lists at most. Since each record holds two integers or one real number, the header holds 8 integers:

1. NP: is the number of vertices the object has.
2. NF: is the number of faces of the object.
3. NFF: is the number of flat faces of the object.
4. NSE: is the number of straight edges of the object.
5. NCE: is the number of curved edges.
6. NEL: is the length of the edge list which is described below.
7. NL: the number of groups of L-numbers described in Braid's work [1]
8. NPL: the number of groups of the limits of parameter t.

The data block is recognised by its (starting) record number.



Complete Picture of a data block representing one object.

I. LIST 1: THE VERTEX LIST

The vertex list holds the three co-ordinates of each vertex of the object. Hence the list contains sequentially elements of the array P(3,NP) where

P(N,I) for N = 1, 2, 3 are the x, y, z co-ordinates of vertex I which is from 1 to NP.

The length of the vertex list is therefore 3 x NP records because each co-ordinate is a real number.

II. LIST 2: THE FACE STATUS LIST

The face status list contains all integers, each of which is an element of the array FS(3,NF) where

FS(N,I) for N=1, 2, 3 are the 3 integer status of face I which is an integer and has values of 1 to NF.

The three integers indicating the status of each face I are:

1. FS(1,I) = f_1

This integer shows the type of the face I.

If face I is positive, f_1 is a positive number and if face I is negative, f_1 is a negative number.

If $|f_1| \varepsilon (1,100)$, face I is a flat face whose face equation is held in the $|f_1|$-th column of the array AFF(4,NFF) described in list 3 below.

If $|f_1| \varepsilon (101,200)$, face I is a cylindrical face whose parametric equation is held in the $(|f_1|-100)$-th entry of the array AFC(5,3,NAFCD) described in list 4.

If $|f_1| \varepsilon (201,300)$, face I is a conical face whose parametric equation is held in the $(|f_1|-200)$-th entry of the array AFC(5,3,NAFCD).

With this convection, any object is allowed to have a maximum of 100 flat faces and/or 100 cylindrical faces and/or 100 conical faces! which is found quite sufficient in practice.

2. FS(2,I) = f_2

f_2 is a flag which when it is set to 1 indicates that face I has been deleted, e.g. during merging algorithm. Otherwise f_2 is equal to zero.

3. FS(3,I) = f_3

f_3 holds the start of an entry in the edge list EL(NEL) described in list 5, which in turn holds all the edge numbers of the edges belonging to face I.

The length of list 2 is therefore:

= 3 x NF/2 records if NF is even

= [(3 x NF) + 1]/2 records if NF is odd.

III. LIST 3: THE FLAT FACE EQUATION LIST

This list holds all elements of the array AFF(4,NFF), each of which is a real number.

Each column I of this array holds 4 numbers A, B, C, D of the equation of the flat face I:

$$Ax + By + Cz + D = 0$$

where I is an integer and has values from 1 to NFF.

Note: A, B, C are the direction cosines of the unit normal outwards from the face (for a positive face) while D is the signed length of the vector normal to the face and from the face to the origin. The positive direction is the direction of the normal vector of the face.

The length of list 3 is therefore 4 x NFF records.

IV. LIST 4: THE PARAMETRIC EQUATION LIST

This list holds the elements of the parametric equations of all curved faces of the object. Of course when the object does not have any curved surfaces, signalled by NF = NFF, this list is not present in the data structure.

The parametric equation of either cylindrical faces or conical faces is shown by Braid [1] to be a 5 x 3 matrix. Hence each entry in

this list contains 15 real numbers. For example, the 5 x 3 matrix equation of curvedface I is held in the array AFC (N1,N2,I) where

N_1 is an integer and has values from 1 to 5

N_2 is an integer and has values from 1 to 3

I is an integer and has values from 1 to NAFCD

where NAFCD = NF - NFF and NF>NFF.

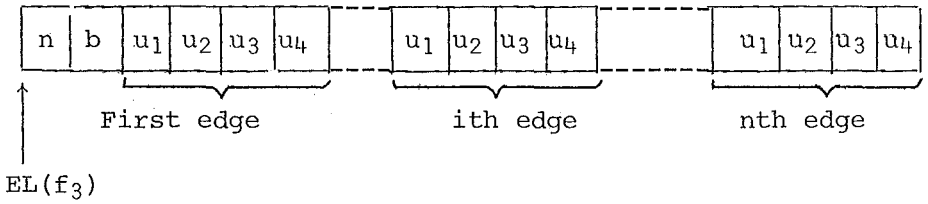
The length of list 4, if present, is 15 x NAFCD records.

V. LIST 5: THE EDGE LIST

List 5 contains all elements (integers) of the edge list array EL(NEL) where NEL is the length of the array. Array EL contains NF entries. Each entry holds data of all the edges belonging to a particular face. The start of each entry is held in the array FS, being f_3.

The structure of array EL is shown below:

Complete EL array



Structure of the I-th entry

The structure of each entry is as follows:

1. n: is the first number of the entry, held in EL (f_3)

   n is the total number of all edges lying on face I.

2. b: is a flag = 0: surface I has no holes.

   = 1: surface I has holes.

3. u_1: is a flag = 0: the current edge belongs to the current loop.

   = 1: the current edge starts a new hole.

For the first edge in each entry u_1 is always 0 because the first edge in each entry is always the starting edge of the outer perimeter

of the face.

4. u_2: gives the type of the edge.

   u_2 = 1: the current edge is straight

   = 2: the current edge is circular

   = 3: the current edge is conic

5. u_3: gives the direction of the edge

   u_3 = 0: the direction of the current edge, whether in straight edge list (list 6) or curvededge list (list 7), is from the vertex 1 to vertex 2 as held in the corresponding list

   u_3 = 1: the direction of the current edge is reverse, i.e. from vertex 2 to vertex 1 as held in the corresponding list.

6. u_4: gives the edge number of the current edge.

Therefore, for each entry n and b gives information about the face and each quadruple (u_1, u_2, u_3, u_4) gives information about each edge of the face. The edges which appear in each entry are in a strict order. If one follows the edges in the order of their appearance, a loop would be traced out in anti-clockwise direction if it is the outer perimeter of the face and in clockwise direction if it is a hole in the face, when looking towards the face against the direction of the normal vector of the face. Since each edge lies on two faces, each edge should appear twice in EL list, one in a forward direction (u_3 = 0) and one in a reverse direction (u_3 = 1).

The length of EL array is therefore:

$$NEL = \sum_{i=1}^{NF} [2 + (n_i \times 4)] \text{ words or NEL/2 records}$$

where i = integer

n_i = number of edges on face i.

VI. LIST 6: THE STRAIGHT EDGE LIST.

This list contains all integer elements of array EVS(4,NSE). Each column I, say, contains 4 numbers EVS(N,I), N = 1, 2, 3, 4 which give information about straight edge I where:

1. e_1 = EVS(1,I): vertex number of vertex 1 of the edge
2. e_2 = EVS(2,I): vertex number of vertex 2 of the edge
3. e_3 = EVS(3,I): is the visibility flag during display

   e_3 = 0: the edge is visible

   = 1: the edge is invisible.

   e_3 is the replacement flag during merging and intersection.

   e_3 = 0: the edge is not replaced by any other edge

   e_3 = 1: the number is replaced by another edge whose number is given by e_4.
4. e_4 = EVS(4,I): the bisecting flag

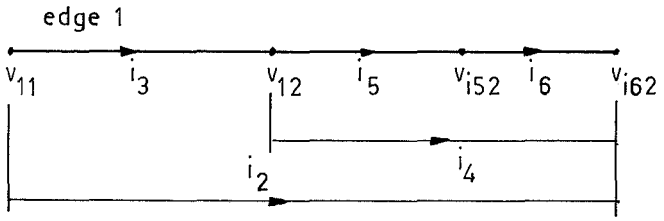
   e_4 = 0: the edge is not bisected

   e_4 = 1: the edge is bisected into 2 other edges whose numbers are given by e_1 and e_2 and in the order from e_1 to e_2.

When e_3 = 1, e_4 contains the edge number of the edge which replaces the current edge I.

When e_3 = 1 and e_4 = 0, the current edge I is deleted completely and not replaced by any other edge.

Edge	e_1	e_2	e_3	e_4
1	v_{11}	v_{12}	0	0
‖	‖	‖	‖	‖
i_1	v_{i11}	v_{i12}	1	0
‖	‖	‖	‖	‖
i_2	i_3	i_4	0	1
‖	‖	‖	‖	‖
i_3	v_{i31}	v_{i32}	1	1
‖	‖	‖	‖	‖
i_4	i_5	i_6	0	1
‖	‖	‖	‖	‖
i_5	v_{12}	v_{i52}	0	0
‖	‖	‖	‖	‖
i_6	v_{i52}	v_{i62}	0	0
‖	‖	‖	‖	‖



An example of the straight edge list during merging.

Thus in our example shown above, edge 1 exists and goes from vertex V_{11} to vertex V_{12}. Edge e_1 is totally deleted. Edge i_3 is deleted and replaced by edge 1. Edge i_4 is bisected into edge i_5 and i_6. And finally edge i_2 is trisected into 3 edges: edge 1, i_5, i_6.

A vertex always lies on three different edges and hence each vertex should appear three times in the straight edge list (or the curved edge list).

The length of the EVS list is 2 x NSE records.

VII. LIST 7: THE CURVED EDGE LIST

Again this list, as the name suggests, will not be present in the data structure when NCE = 0. When NCE>0, this list holds all integer elements of the array EVC(7,NCE). It is similar to EVS list, except that this time each column I has 7 integers giving all information about the curved edge I: the first 2 integers ec_1 = EVC(1,I), ec_2 = EVC(2,I) are exactly the same as e_1 and e_2 respectively in the EVS list. The last two integers ec_6 = EV6(6,I), ec_7 = EVC(7,I) are exactly the same as e_3 and e_4 respectively in the EVS list. The other integers have the following meaning:

1. ec_3 = EVC(3,I).

   ec_3 is the pointer pointing to the L-number group in the array L (list 8) which the current curved edge I is associated with.

2. ec_4 = EVC(4,I): the parameter flag.

   ec_4= 0: the t-parameter limits for the curved edge are 0. and 1. (t varies from 0. to 1.)

   ec_4= 1: the t-parameter limits are 1. and 0.

   ec_4> 0: ec_4 is now a pointer pointing to the particular t parameter limits contained in the PL(2,NPL) group (list 9)

3. ec_5 = EVC(5,I)

   This is a pointer pointing to the ec_5-th curved face as arranged in the AFC list, on which curved edge I lies. The ec_5 pointer is provided to speed up any computation involving the curved edge. Without this pointer, one would have to search every curved face through list FS, EL and test to see which one holds the current edge.

   The length of the EVC list is:

7 x NCE/2 records if NCE is even

[(7 x NCE) + 1]/2 records if NCE is odd.

VIII. LIST 8. THE L-NUMBER LIST

This list holds the L-numbers (real numbers) associated with curved edges. Again when NL = 0, this list is not present in the data structure. When NL>0, this list holds all elements of the array L(6,NL).

When the associated edge I is a circular edge, only the first 4 numbers in column N of array L are accessed, i.e . L_1 = L(1,N), L_2 = L(2,N), L_3 = L(3,N), L_4 = L(4,N) because circular edges are associated to L-number groups of 4 numbers, where N = ec_3 in list 7.

When the associated edge I is a conical edge, all the 6 numbers in column N of array L are considered because conical edges are associated with 6 L number groups. (See Braid [1]).

The length of this list, when present, is (6 x NL) records.

IX. LIST 9: THE PARAMETER LIMIT LIST

This list, when present, holds the lower and upper limits for parameter t of a curved edge. When NPL = 0 this list is not present. When NPL>0, this list holds members (real numbers) of the array PL(2,NPL) where:

1. t_1 = PL(1,N) is the lower limit
2. t_2 = PL(2,N) is the upper limit

of the parameter t associated with curved edge I where N = ec_4>0 in list 7. When t varies from t_1 to t_2 , a point will trace out curved edge I in the right direction.

The length of this list, when present, is (2 x NPL) records.

COMMENTS

1. This data structure is basically a tree structure with one way pointers. This means that some searching processes might be long and inconvenient. For example, to find out which faces hold

a particular vertex, one would have to search through all faces, starting from FS list, to EVS list and/or EVC list if necessary and test to see if each face holds that vertex. On the other hand, if a two-way or three-way pointer system is implemented, the penalty would be, of course, a bigger data structure which requires more core storage. However, such searching processes like surface-vertex search are very rarely performed in this application and therefore this data structure is quite sufficiently efficient.

2. This form of data structure has some degree of redundancy. For example, a box (or actually a parallelepiped) may be completely defined by giving the co-ordinates of four vertices, not 8. Another example in the storing of end vertices of curved edges. When parameter t has values at its upper or lower limits, these end vertices will be found and hence defined once more. Another example is the storing of flat face equations. Since any 3 non-colinear vertices define a flat face, they can be used to yield the face equation.

   However, the point here is that a uniform data structure is required for all shapes of objects and so is a reasonable degree of efficiency and when these requirements are met, redundancy results.

3. When implemented in a program, a dimension declaration such as EVC(7,NCE) can generate a mistake when NCE = 0. To get away with this, 4 complementary numbers are formed as below:

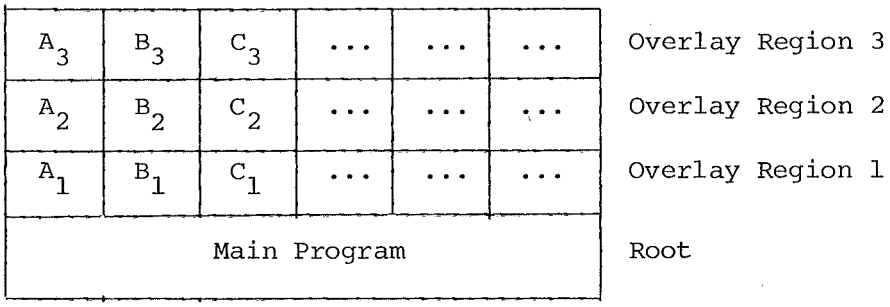
AFC(5,3,NAFCD)	NAFCD		1		NF		NFF
EVC(7,NEVCD)	NEVCD		1		NCE		0
L(6,NLD)	NLD	=	1	when	NL	=	0
PL(2,NPLD)	NPLD		1		NPL		0

Otherwise, they are equal to (NF-NFF), NCE, NL, NPL respectively.

APPENDIX C.

PROGRAM OVERLAY STRUCTURES AND COMMANDS

The programs described in chapter 6 were all written in FORTRAN and in an overlay environment. Details of various overlay regions and overlay segments for each program are given here. In these programs, the arrangement of overlays is determined by the order in which overlaid subroutines are called rather than by the sizes of the subroutines. This is one characteristic of an overlay structure run under the PDP-11 system [19]. Briefly, a routine in an overlay segment can call another routine in the same segment or in the root or in a segment in another higher-numbered overlay region.



A_1, B_1, C_1, etc.: overlay segments sharing overlay region 1.
Only one of these segments is resident in core at any instance of time.

Root: is the part which is always resident in core. It is usually the main program controlling other overlays.

Fig. C.1 The Overlay Structure

Thus in Fig. C.1 a routine in A_2, say can call another routine in the main program or within A_2 or in A_3 or B_3 or C_3 etc. It cannot call another routine in B_2 or C_2 or A_1 or B_1 or C_1, etc.

All commands have three alphabetic letters and can be entered at different parts of a program. When the program is ready to receive a command, one or more asterisks (*) is typed out on the decwriter. One

asterisk means that control is at the root, two means control is in a segment in the overlay region 1, and so on. Sometimes a message is typed out asking for a command or data and the user is required to enter accordingly. All integers and real numbers can be entered in free format forms. Consecutive numbers can be separated by commas. If a <CR> (carriage return) key is typed in in response to an *, the current stage is completed and the program proceeds to the next stage.

In the following description a { } indicates what the user enters in response to a program request, followed by a <CR> key.

I. TGDEF:

This program defines primitive objects. Its structure is shown in Fig. C.2.

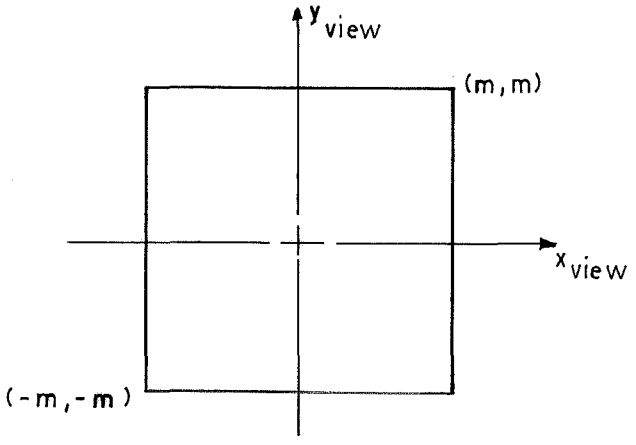
1. The Root: The root is approximately 3.4K words long and 3K of which is reserved for the display buffer.

At the start of the execution, the root asks for:

(i) The scale parameter by typing out the message:

ENTER SCALE PARMTR: {m}

The scale parameter defines the viewing window of the screen. When a number m is entered in response to this message, the screen is defined to be a square window with its bottom left hand corner having the co-ordinates (-m, -m) and its top right hand corner having the co-ordinates (m, m).



	APPROXIMATE SIZES (K)											
O/L REGION 6	-	-	-	-	-	-	-	0.2	-	-	-	-
O/L REGION 5	0.1	-	0.4	0.6	0.1	-	-	0.9	0.2	-	-	-
O/L REGION 4	0.7	-	0.6	0.3	0.6	-	0.1	0.6	0.3	0.1	-	-
O/L REGION 3	0.7	0.6	0.7	0.7	0.6	0.6	0.6	1.1	0.5	0.6	-	-
O/L REGION 2	1.9	0.3	0.8	1.4	0.6	0.5	0.3	0.7	0.6	0.8	0.2	-
O/L REGION 1	0.8	2.0	1.1	0.9	1.2	0.7	2.2	0.7	2.2	1.7	0.7	0.5
ROOT	3.4	4.2	4.8	5.3	4.7	4.6	4.2	5.1	5.2	7.0	5.1	5.5
TOTAL	7.6	7.1	8.4	9.2	7.8	6.4	7.4	9.3	9.0	10.2	6.0	6.0
	TGDEF	TGMER1	TGME21	TGME22	TGMER3	TGMER4	TGINT1	TGINT2	TGINT3	TGDESN	TGNEG	TGDIS

Fig. C.2. THE APPROXIMATE SIZES OF THE PROGRAMS

In other words, when an object is transformed from its object co-ordinate system into the viewing co-ordinate system, only those points having $x_{view} \varepsilon(-m,m)$ and $y_{view} \varepsilon(-m,m)$ can be displayed on the screen. Also the size of the screen is 2 m x 2 m unit lengths.

(ii) The view point by typing out the message:

ENTER VIEW POINT: {x, y, z}

The displayed view from this time onwards will be as seen in the direction of the viewing vector. The viewing vector is defined to be the vector which starts at the view point and ends at the origin.

The root then types out an * indicating that it is now at its ready state i.e. that it is ready to accept a command from the user. The user can enter any one of the commands shown below. At the end of each command execution, control goes back to the ready state in the root. However, when a FIN command is entered, the program is terminated.

(a) VIW: to re-define the view point. The currently-displayed view is deleted and a new view corresponding to the new view point is displayed.

(b) DEF: to call a routine to define a positive primitive object.

(c) SCR: to call a routine to turn on, turn off or delete a sub-picture.

(d) HED: to output the object headers (i.e. the defining parameters) of all objects currently in the disc file.

(e) NEG: to negate a primitive object.

(f) FIN: to terminate the execution of the program.

2. Overlay Region 1: contains various routines which can be initiated by a command in the root. There are 5 overlay segments in region 1:

(a) Segment 1: generates a new view on the screen when the three co-ordinates of a new view point are entered.

(b) Segment 2: defines a primitive object. When this segment is loaded into core and run, '**' is printed on the terminal asking for the name of the primitive object to be defined. The name can be any one of the six names:
BOX, CYL, WED, FIL, CON, and PYR.
After one object is defined, control goes back to the ready state, i.e. '**' is typed on the terminal. If a <CR> key is typed in, control exits from segment 2 and goes back to the root.

(c) Segment 3: alters the current display on the screen. Again a '**' is printed asking for a further command. The commands in this segment are:

n, OFF: n is an integer. Subpicture n is temporarily turned off from the display.

n, ON: subpicture n, which has previously been turned off is displayed again.

n, DEL: subpicture n is completely and permanently deleted from the display and the data file.

Any attempt to turn on, turn off or delete a subpicture which has already been turned on, turned off or deleted respectively is ignored.

(d) Segment 4: reads the object headers off all currently-created primitive objects and outputs them on the terminal.

(e) Segment 5: negates a primitive object. The data structures of the negated object and the original object are stored separately. This segment reads the object number (entered by the user), calls a routine to negate it and stores it on the file.

3. Overlay Region 2: contains 8 segments:

(a) Segment 1: generates and stores and calls a routine to display a primitive BOX when its defining parameters are entered.

(b) Segment 2: does exactly the same as segment 1 except that the primitive object is a CYL.

(c) Segment 3: the same as segment 1 but the primitive object is a WED.

(d) Segment 4: same as segment 1 but the primitive is a FIL.

(e) Segment 5: same as segment 1 but the primitive is a CON.

(f) Segment 6: same as segment 1 but the primitive is a PYR.

(g) Segment 7: negates a given data structure of an object. This segment is initiated by segment 5 of O/L region 1.

(h) Segment 8: turns off all the currently-displayed subpictures and displays them again under a new viewing vector given by segment 1 of region 1.

4. Overlay Region 3: contains 8 segments:

(a) Segment 1: displays a wire-frame picture of an object.

(b) Segment 2: inverts a square matrix.

(c) Segment 3: reverses the direction of the edges around a loop (required during object negation).

(d) Segment 4: outputs a data structure to the disc file.

(e) Segment 5: inputs a data structure from the disc file.

(f) Segment 6: creating a viewing matrix (transformation matrix) given a view point.

(g) Segment 7: outputs the individual object header from the disc file to the terminal.

(h) Segment 8: outputs the object header to the disc file as an object is defined.

5. Overlay Region 4: contains 2 segments:

(a) Segment 1: contains various small routines to:

- calculate the value of parameter s, given t, and the corresponding co-ordinates of a point on a cylindrical edge.
- calculate the value of parameter s, given t, and the corres-

ponding co-ordinates of a point on a conical edge.

- re-initialize a matrix i.e. set all elements to 0.
- set a matrix to an identity matrix.
- create a rotation matrix given the axis of rotation (OX, OY, or OZ) and angle of rotation.

(b) Segment 2: contains small routines to:

- print out the complete data structure of an object onto the terminal.
- input or output an array of integers to or from the disc file.
- input or output an array of real numbers to or from the disc file.

6. Overlay region 5: contains one single segment to multiply two matrices. This segment has to be put in overlay region 5 by itself to allow it to be called by other routines in lower-numbered overlay regions.

This program is comparatively short, its total length is approximately 7.6 K, well below the available size of approximately 11 K (Refer to Chapter 6).

II. TGMER1: does the first stage of merging. The structure is shown in Fig. C.2.

1. The Root: is approximately 4.2 K words long. The common block reserved for the data structure alone takes 3.6 K.

At execution, the root will type out a message asking for the record numbers of object 2 and object 1 (object 2 is merged into object 1). Then it calls a routine to do stage 1 of merging. At the end of stage 1, it outputs the data structure as it is to the reserved area on the disc file and the program stops.

2. Overlay Region 1: has only one single segment which does the first stage of merging.

Initially the overlay region 1 was intended to have 4 overlay

segments, each segment does one stage of merging. When it was later found out that stage two was too big to fit in, the overlay region was left as it was. Because this segment is the only one in region 1, it could have been put in the root but either way would require the same storage space in core.

3. Overlay Region 2: has 3 segments.

(a) Segment 1: transfers the data structure of an object from the disc file to the core.

(b) Segment 2: transfers the data structure of an object from the core to the disc file.

(c) Segment 3: concatenates two integer lists or two real number lists, one after the other.

4. Overlay Region 3: has 4 segments, many of which are exactly the same as mentioned in TGDEF.

(a) Segment 1: transfers a data list of integers or real numbers from the disc file to the core.

(b) Segment 2: transfers a data list of integers or real numbers from the core to the disc file.

(c) Segment 3: types out on the terminal the complete data structure of the object.

(d) Segment 4: has many small routines to:

- multiply two matrices
- normalize the flat face equation
- initialize a matrix of integers or real numbers
- make an identity matrix
- push a stack down the list to make room for inserting more numbers.

This program is also short having a total length of about 7.1 K.

III. TGME21: does the first half of the second stage of merging. The structure is shown in Fig. C.2.

1. The Root. The root is approximately 4.8 K words long and as TGMER1, 3.6 K words are reserved for the data block.

At the start of the execution, the root reads the data structure from the reserved area on the disc file into its data block. It then looks for all pairs of flat coplanar faces and for each pair it calls a routine to check if the pair can be merged. For those that can, it calls another routine to do the first half of the second stage. Finally it dumps the data structure back on the reserved area on the disc file and stops.

2. Overlay Region 1: has 3 segments.

(a) Segment 1: tests to see if the outer perimeters of two co-planar faces are disjoint. If they are, they cannot be merged and the pair is ignored.

(b) Segment 2: does the first half of the second stage, i.e. finds all intersections between straight edges of the coplanar faces.

(c) Segment 3: accommodates many small I/O routines to:

- read a data structure from the disc file.
- write a data structure onto the disc file
- read a sequential list of integers or real numbers from the disc file
- write a sequential list of integers or real numbers onto the disc file.

3. Overlay Region 2: has 4 segments.

(a) Segment 1: has two routines:

- to inspect the relative position of two loops.
- to find the intersection of two straight line segments (Appendix E)

(b) Segment 2: splits intersecting edges and stores the intersection.

(c) Segment 3: splits and stores two overlapping straight edges.

(d) Segment 4: tests to see if two meeting faces can be combined and if they can, calls routines to combine them.

4. Overlay Region 3: has 4 segments.

(a) Segment 1 and Segment 2: deals with different cases of combining 2 meeting faces.

(b) Segment 3: has 3 routines to:

- make a separate list of all the edges and its pieces belonging to a face
- mark a face as being deleted.
- add a new face to the data structure.

(c) Segment 4: deals with the different cases of splitting overlapping edges.

5. Overlay Region 4: has 8 segments:

(a) Segment 1: has routines to

- bisect an edge
- find and store in a list all pieces of a split edge,

(b) Segment 2: has 2 routines to

- copy edges from a simple edge list to a face being made without recording the traced area until a special vertex is met
- add a complete loop from a simple edge list to a face being made

(c) Segment 3: has 2 routines to:

- find a relative position between an edge and a point lying on the line containing the edge
- subroutine INSPEC (see Chapter 4).

(d) Segment 4: has 2 routines to

- find the face which meets a given face at a given edge
- find the change in slope between two meeting faces across their

common edge

(e) Segment 5: deletes an edge from the structure.

(f) Segment 6: deletes a vertex from the structure.

(g) Segment 7: has many small routines to

- create a new vertex
- find the cross product of 2 vectors
- concatenate 2 lists of integers or real numbers
- move the stack up a sequential list of integers or real numbers to delete a given number of elements in the list.

(h) Segment 8: copies edges from a simple edge list to a face being made until a special vertex is met and at the same time records the area being traced.

6. Overlay Region 5: has 3 segments

(a) Segment 1: has many small routines to

- find all pieces of a split edge
- push the stack down a list
- initialize a list.

(b) Segment 2: has routines to

- find the dot product of two vectors
- initialize a matrix
- make an identity matrix.

(c) Segment 3: has 2 routines to

- find a suitable projection plane for a face
- reverse the direction of an edge and its pieces in the edge-vertex list.

The total length of this program is 8.4 K words long.

IV. TGME22 does the second half of the second stage of merging. The structure is shown in Fig. C.2.

1. The Root: The size of the root is approximately 5.3 K and again

3.6 K words are reserved for the data block.

TGME22 completes the second stage of merging by finding all intersections between straight-curved edges and curved-curved edges. The finding of these intersections often involves iterations and hence takes time especially when curved edges do not intersect at all. Because of this, the user has an opportunity to by-pass this program if it is not necessary.

At the start of the execution, a message "CURVED EDGE INTERSECTION?:" is typed out on the terminal. If the user types a <CR> key in response to this message, the program will be terminated. If he types any non-zero character, the execution of TGME22 commences. It reads the data structure from the reserved area on the disc file into core and for each pair of coplanar faces (already found by TGME21), it looks for all pairs of straight-curved edges and curved-curved edges, calls routines to find their intersections. When all have been found, it dumps the data structure back on the reserved area on the disc file.

2. Overlay Region 1: has 3 segments

   (a) Segment 1: deals with a pair of curved edges. A quick test using enclosing triangles (Chapter 3) is used to eliminate non-intersecting edges. If the pair passes this test, the segment calls various other routines to determine its intersection.

   (b) Segment 2: has 2 routines

   - one is very similar to segment 1. The pair consists of one straight edge and one curved edge
   - the other routine obtains a list of all pieces of a straight, split edge.

   (c) Segment 3 - splits a straight edge at one or two inter-section points.

3. Overlay Region 2: has 7 segments

(a) Segment 1: has many small routines to

- create a new vertex

- calculate the values of parameters corresponding to a given point lying on a cylindrical or a conical edge.

(b) Segment 2: has 2 routines to

- bisect a straight edge

- bisect a curved edge

(c) Segment 3: finds the intersections of two cylindrical - cylindrical edges.

(d) Segment 4: finds the intersections of a pair of cylindrical - straight edges.

(e) Segment 5: finds the intersections of a pair of conical - conical edges.

(f) Segment 6: finds the intersections of a pair of conical - straight edges.

(g) Segment 7: finds the intersections of a pair of cylindrical - conical edges.

4. Overlay Region 3: has 5 segments

Segment 1 has 2 routines to

- solve the set of parametric equations for intersections between cylindrical and straight edges

- delete a vertex from the data structure.

Segments 2, 3, 4 and 5 solve the sets of parametric equations for finding intersections between cylindrical - cylindrical, cylindrical - conical, conical - conical, conical - straight edges using Brown's iteration routine [4].

5. Overlay Region 4: has 8 segments

(a) Segment 1: forms a list of pieces of a split, curved edge from the data structure

(b) Segment 2: has 2 routines to:

- reverse the direction of a curved edge and its pieces in the curved edge - vertex list

- finds the numerical values of functions describing the intersections of cylindrical-cylindrical edges for use with Brown's iteration routine

(c) Segment 3: has 3 routines to find the numerical values of the functions describing the intersections of cylindrical - conical, conical - conical, conical - straight edges

(d) Segment 4 - writes the whole data structure on the disc file

(e) Segment 5: reads the whole data structure from the disc file

(f) Segment 6: has 2 routines to

- calculate the tangent - crossing point.

- determine the relative position between an edge and a triangle (Appendix D)

(g) Segment 7: has 2 routines to

- determine the relative position between an edge and a point lying on its line

- calculate the co-ordinates of a point lying on a cylindrical edge given the values of its parameters

(h) Segment 8: calculates the co-ordinates of a point lying on a conical edge given the values of its parameters.

6. Overlay Region 5: has 3 segments

(a) Segment 1: has many routines to read or write a sequential list of integers or real numbers from or to the disc file. It also contains routines to find the dot product and the vector product of two vectors

(b) Segment 2: has many routines to

- find the value of a polynomial

- push down the stack of a sequential list

- initialize a list

- find a suitable projection plane

- multiply two matrices

(c) Segment 3: has 2 routines

- INSPEC

- determine if a parameter lies outside its allowed limits

The total length of TGME22 is approximately 9.2 K.

V. TGMER3: does the third stage of merging. Its structure is shown in Fig. C.2.

1. The Root: The root is approximately 4.7 K long and as before 3.6 K are reserved for the data block.

The root reads the intermediate data structure from the disc file and for each pair of flat coplanar faces, it calls a routine to assemble them (stage 3). Then it writes the structure back on the disc file.

2. Overlay Region 1: has only one single routine controlling the assembly of edges to form new faces.

3. Overlay Region 2: has 4 segments

(a) Segment 1: has 2 routines to

- write a complete data structure onto the disc file

- re-assemble a pair of faces having intersecting perimeters

(b) Segment 2: re-assembles a pair of faces when the perimeter of one face lies inside that of the other

(c) Segment 3: re-assembles intersecting holes of the faces

(d) Segment 4: reads a complete data structure from the disc file into core.

4. Overlay Region 3: has 4 segments

(a) Segment 1: makes a simple list of edges and their pieces belonging to a face

(b) Segment 2: has 2 routines to

- add an edge from a simple edge list to the face being made, at

the same time recording the traced area until a special vertex or the initial vertex is met.

- add a new face to the current structure.

(c) Segment 3: has 4 routines to

- mark a face as being deleted
- determine the relative position of two perimeters
- add a loop from a simple edge list to a new face being made as a hole or as the perimeter of the new face
- make a list of vertices lying on a loop

(d) Segment 4: finds an un-used hole, without intersections from a simple edge list, which lies inside a new face and adds it to the new face

5. Overlay Region 4: has 2 segments

(a) Segment 1: has many routines to

- make a list of pieces of a split edge (straight or curved)
- concatenate 2 sequential lists of integers or real numbers
- move up the stack of a sequential list

(b) Segment 2: has many routines to read and write a sequential list of real numbers or integers from or to the disc file

(c) Segment 3: has routine INSPEC.

6. Overlay Region 5: has 2 segments

(a) Segment 1: has 3 routines to

- push down the stack of a sequential list
- initialize a list
- find a suitable projection plane

(b) Segment 2: has 2 routines to

- initialize a matrix
- make an identity matrix.

The total length of TGMER3 is approximately 7.8 K.

VI. TGMER4: does the fourth stage of merging. The structure is shown in Fig. C.2.

1. The Root: The root is 4.6 K long and again 3.6 K of which is reserved for the data block.

The root reads the data structure of the object on the disc file and then calls a routine to tidy up the data structure (stage 4). It then types out the message:

ENTER REC. NO. TO BE WRITTEN ON:

and the number entered is the number of the starting record of the data block on which the data structure is written.

2. Overlay Region 1: has one single segment to tidy up the data structure (stage 4).

3. Overlay Region 2: has 9 segments

(a) Segment 1: has routines to find out all edges and vertices which no longer belong to any faces and edges respectively and deletes them

(b) Segment 2: has routines to find out all L-number groups and pairs of parameter limits which no longer correspond to any curved edges and deletes them

(c) Segment 3: re-numbers the curved edge - vertex list, deleting those marked as deleted and replacing split edges by their pieces

(d) Segment 4: is the same as segment 3 except that the list is the straight edge - vertex list

(e) Segment 5: forms a list of non-deleted pieces of a straight or a curved edge which has been split

(f) Segment 6: deletes all faces which have been marked as deleted

(g) Segment 7: reads a complete data structure from the disc file

(h) Segment 8: writes a complete data structure to the disc file

(i) Segment 9: looks for common vertices which become redundant when two straight colinear edges join together and deletes them.

4. Overlay Region 3: has 2 segments

(a) Segment 1: deletes all vertices which no longer belong to any existing edges

(b) Segment 2: has many routines to

- push down the stack of a sequential list
- move the stack up a sequential list
- read or write a sequential list of integers or real numbers from or to the disc file
- initialize a matrix

This is the shortest program having a length of 6.4 K.

It is worth noting that all five programs of merging algorithm are each smaller than the available amount of core but they would be too big if they were put together. However, thanks to this amount of free core that BATCH can be loaded to run the five programs as a batch job.

To run merging (TGMERG) under BATCH, it is necessary to assign I/O devices and load them, together with BATCH [19]. These can be done by the following sequence of monitor commands:

```
.ASS TT:LOG
.ASS TT:LST
.LOAD BA,TT
.R BATCH
*<filename> (/X) <CR>
```

In this case the I/O device is the dec-writer. The file name to be input after the * would be TGMERG. The switch /X indicates that the batch stream TGMERG is a pre-compiled stream i.e. it is in a form executable by the BATCH run-time handler. Without this switch, the BATCH compiler will have to compile the source stream before it can be run.

VII. TGINT1: does the first stage of intersection. Its structure is shown in Fig. C.2.

1. The Root: is 4.2 K long and again 3.6 K is reserved for the data block.

The root does exactly the same as the root in TGMER1 except that it calls another routine to do the first stage of intersection.

2. Overlay Region 1: has one single overlay segment which does the first stage of intersection.

3. Overlay Region 2: has 6 segments:

(a) Segment 1: reads a complete data structure from the disc file

(b) Segment 2: writes a complete data structure to the disc file

(c) Segment 3: concatenates 2 sequential lists of integers or real numbers

(d) Segment 4: marks non-intersecting faces as non-intersecting as a result of the sphere test

(e) Segment 5: finds a point lying on a curved face and being nearest to the origin

(f) Segment 6: forms the translational matrix which brings the centre of the sphere enclosing an object to the origin.

4. Overlay Region 3: has 7 segments

(a) Segment 1: has routines to read a sequential list of integers or real numbers from the disc file

(b) Segment 2: has routines to write a sequential list of integers or real numbers to the disc file

(c) Segment 3: normalizes the flat face equations

(d) Segment 4: calculates the co-ordinates of a point lying on a conical face given the values of a pair of parameters

(e) Segment 5: like segment 4 except that the point lies on a cylindrical face

(f) Segment 6: calculates the dot and vector products of two vectors

(g) Segment 7: inverses a matrix

5. Overlay Region 4: has 2 segments

(a) Segment 1: multiplies two matrices

(b) Segment 2: initializes a matrix of integers or real numbers.

The total length of this program is 7.4 K.

VIII. TGINT2: does the second stage of intersection. Its structure is shown in Fig. C.2.

1. The Root: is 5.1 K long and 3.7 K of which is reserved for the data block.

The root reads the data structure of the object from the reserved area on the disc file. It then matches all possible face-edge pairs and calls routines to find their intersections (stage 2). Finally, it writes the data structure back to the reserved area and stops.

2. Overlay Region 1: has 5 segments

(a) Segment 1: reads and writes a complete data structure from and to the disc file

(b) Segment 2: finds the intersection point of a flat-face and a straight edge

(c) Segment 3: finds the intersection point(s) of a flat face and a curved edge

(d) Segment 4: finds the intersection point(s) of a cylindrical face and a straight edge

(e) Segment 5: finds the intersection point(s) of a conical face and a straight edge

3. Overlay Region 2: has 4 segments

(a) Segment 1: finds the numerical solutions for the equations describing the intersection between a flat face and a cylindrical edge by iterations

(b) Segment 2: finds the numerical solutions for the equations describing the intersection between a flat face and a conical edge by iterations

(c) Segment 3: similar to segment 1 except that the intersection is between a cylindrical face and a straight edge

(d) Segment 4: similar to segment 1 except that the intersection is between a conical face and a straight edge

4. Overlay Region 3: has 7 segments

(a) Segment 1: reads and writes a sequential list of integers or real numbers from and to the disc file

(b) Segment 2: has many small routines to

- initialize a matrix
- find a suitable projection plane
- find the next iteration using Newton-Raphson method

(c) Segment 3: has many routines to:

- make a list of vertices forming the perimeter of a face
- test to see if a point is on a vertex of a face
- determine the sign of the dot product of a vector and the normal vector of a flat face

(d) Segment 4: has many routines to

- determine the sign of the dot product of a vector and the normal vector of a curved face
- subroutine CINSPE.

(e) Segment 5: has many routines to:

- split a straight edge and store the information at 1 or 2 intersection points
- find the roots of a quadratic equation

(f) Segment 6: determines if a point lies inside a hole on a face

(g) Segment 7: splits and stores a curved edge at 1 or 2 intersection points.

5. Overlay Region 4: has 4 segments

(a) Segment 1: has many routines to

- create a vertex
- delete a vertex
- store a vertex in a special vertex list

(b) Segment 2: bisects a straight or curved edge

(c) Segment 3: makes a list of all pieces of a split edge (straight or curved)

(d) Segment 4: finds the normal vector to a cylindrical face or a conical face at given parameter values.

6. Overlay Region 5: has 2 segments

(a) Segment 1: has many routines to

- determine the relative position between a point and an edge
- determine if a parameter value lies outside its limits
- push the stack down a sequential list
- initialize a matrix
- find the cross product of two vectors

(b) Segment 2: has routine INSPEC and others to calculate the co-ordinates of a point lying on a cylindrical or a conical face given its parameter values

7. Overlay Region 6: has one single segment to

- multiply two matrices
- find the dot product of 2 vectors
- evaluate a polynomial

The total length of this program is 9.3 K.

IX. TGINT3: does the third stage of intersection. Its structure is shown in Fig. C.2.

1. The Root: is 5.2 K and again 3.7 K is reserved for the data block.

The root reads the intermediate data structure of the object on the reserved area on the disc file. It calls routines to assemble inter-

secting faces and also sorts out which faces should be deleted from the data structure (stage 3). Then it translates the object back to its initial position, writes the data structure back on the reserved area and stops.

2. Overlay Region 1: has 5 segments

   (a) Segment 1: has many routines to

   - read or write a complete data structure from or to the disc file
   - read or write a sequential list of integers or real numbers from or to the disc file

   (b) Segment 2: forms a separate IVEF list involving all intersections of a given face

   (c) Segment 3: assembles a face of the first object

   (d) Segment 4: assembles a face of the second object

   (e) Segment 5: has 3 routines to

   - determine which faces are excluded from the final data structure
   - unmark those faces which have previously been marked as non-intersecting
   - translate the object back to its original position.

3. Overlay Region 2: has 10 segments

   (a) Segment 1: has 2 routines to

   - find a suitable projection plane
   - determine the relative position between a point and an object

   (b) Segment 2: forms a list of all edges and their pieces which lie on a given face

   (c) Segment 3: has 2 routines to:

   - make a list of all vertices which lie on the perimeter of a given face
   - determine if a point is in another list of points

   (d) Segment 4: adds edges from a simple edge list to a new face being made until a special vertex or the initial vertex is met.

While adding edges, the area traced by them is recorded.

(e) Segment 5: has 4 routines to

- obtain a nearest point to a given point from a list of points
- create a straight edge
- obtain the meeting face of a face at a given edge
- add a flat face to the data structure

(f) Segment 6: creates a new curved edge

(g) Segment 7: finds unused, non-intersecting holes from a simple edge list and which lie inside (or outside) a given loop and adds them to the new face being made

(h) Segment 8: has 3 routines to

- add a curved face to the data structure
- add a loop in a simple edge list to the new face being made as a hole or as a perimeter
- reverse the direction of the edges of a hole

(i) Segment 9: has routines to

- form a list of new edges created on a face
- calculate the projected area of a loop on the projection plane

(j) Segment 10: has routines to

- store a loop with its signed area to a list of either holes or perimeters
- find the optimum hole in the list of holes
- find the optimum perimeter in the list of perimeters
- copy part of one sequential list to another list

4. Overlay Region 3: has 7 segments

(a) Segment 1: has routine INSPEC

(b) Segment 2: calculates the co-ordinates of a point lying on a cylindrical face corresponding to given values of parameters

(c) Segment 3: similar to segment 2 except that the point lies on a conical face

(d) Segment 4: calculates the parameter values corresponding to a given point lying on a cylindrical or a conical face

(e) Segment 5: has many different routines to:

- find the dot product of 2 vectors
- find the normal vector to a cylindrical face corresponding to given values of parameters

(f) Segment 6 - finds the normal vector to a conical face corresponding to given values of parameters

(g) Segment 7: has different routines to concatenate 2 sequential lists of integers or real numbers

5. Overlay Region 4: has 3 segments

(a) Segment 1: forms a list of all pieces of a split edge (straight or curved)

(b) Segment 2: has various routines to

- multiply two matrices
- find the cross product of two vectors
- determine if a parameter value is outside its limits

(c) Segment 3: is an iteration routine based on the Newton-Raphson method for finding parameter values corresponding to a point lying on a conical face

6. Overlay Region 5: has one single overlay segment having various small routines to:

- evaluate a polynomial
- initialize a matrix of integers or real members
- push the stack down a list

The total length is approximately 9 K.

The intersection algorithm can be run under BATCH in a similar manner as the merging algorithm. The batch stream file name is TGINTN.

X. TGDESN:

This is a program to assist the user to design a general die assembly. Its structure is shown in Fig. C.2.

1. The Root

The root is the biggest block in the program. It is approximately 7 K long, 3 K of which is reserved for the data block and another 3K is reserved for the display buffer. The actual programming in the root is approximately 1 K.

The root controls the flow of the design process, allowing for the user's interactions at various steps as described in chapter 1 by typing out the corresponding messages.

In the present program, only compression process and a single-cavity die assembly are implemented. The 20 ton Bipel press is the only available machine.

2. Overlay Region 1: has 5 segments

(a) Segment 1: has 2 routines to

- form a viewing matrix
- generate a BOX insert and its negative, expanded shape

(b) Segment 2: generates a PIN insert and its negative, expanded shape

(c) Segment 3: has 3 routines to

- fetch and display the product, store the object numbers of the primitive objects that constitute the product
- fetch and display the 20 ton Bipel press
- rotate the cavity and its constituent objects with the cavity reference point remaining stationary.

(d) Segment 4: translates the cavity and its constituent objects to the required depth

(e) Segment 5: fetches and displays the return pins and the pressure pads for a particular mode of ejection

(f) Segment 6: generates the ejector pins and their negative, expanded shapes

3. Overlay Region 2: has 5 segments

(a) Segment 1: generates a wire-frame picture of an object

(b) Segment 2: generates a negative pin from a positive pin, making allowance for clearance

(c) Segment 3: modifies the cavity and its constituent objects when a change in its position or orientation is made

(d) Segment 4: has 2 routines to

- scale up an object
- negate an object

(e) Segment 5: translates an object along a given vector

4. Overlay Region 3: has 6 segments

(a) Segment 1: inverses a matrix

(b) Segment 2: modifies the data structure of an object under a given transformation

(c) Segment 3: forms a rotation matrix given the axis and the angle of rotation

(d) Segment 4: has routines to:

- calculate the L-number group of a cylindrical edge resulting from the intersection of a cylindrical face and a flat face
- reverse the direction of the edges around a loop

(e) Segment 5: has routines to read

- a complete data structure from the disc file
- a sequential list from the disc file

(f) Segment 6: has routines to write

- a complete data structure to the disc file
- a sequential list to the disc file

5. Overlay Region 4: has 2 segments

(a) Segment 1: multiplies two matrices

(b) Segment 2: initializes a matrix

This is the longest program. Its length is approximately 10.2 K, which is very near to the maximum allowable size.

XI. TGNEG: negates a general object. This program is exactly the same as the one implemented in TGDEF or TGDESN, except that the data block for holding the data structure is much bigger (~ 4 K) to accommodate more general objects. Its total length is about 6 K.

At execution time, it asks for the record number of the starting record of the data block holding an object to be negated. After the negation, it asks for the record number of the starting record of another data block to hold the negated structure. It then goes back to the beginning, asking for more objects to be negated. When there are no more objects to be negated, a <CR> is entered and the program is terminated.

XII. TGDIS: displays the wire-frame picture of an object. It also makes and stores the display file of that picture so that a hard copy may be produced. The total length is about 6 K.

At execution time, it asks for the scale parameter and the position of the view point as in TGDEF, and then the record number of the starting record of the data block holding the object to be displayed. At the end of each display, it types:

SAVE : : {m}

If m, an integer, is entered, the current display is saved in a file called PICm.DPY, by default. If a <CR> is entered, the display is not saved. It then goes back, asking for more objects to be displayed. When there are no more objects, a <CR> is entered to terminate the program.

To get a hard copy, the display file (the file with an extension .DPY) must be converted to a format suitable for the system plotter.

This can be done by running a system program, called DPLOT, as follows:

```
. R DPLOT
* filename .PLT = filename .DPY <CR>
```

where the filename .PLT is the file ready to be plotted.

APPENDIX D

A SUMMARY OF THE REPRESENTATION OF FACES AND EDGES AND INTERSECTION FORMULAE

A detailed discussion about representing faces and edges is given in Braid's work [1]. A summary of the forms used in this work is given here.

1. Flat faces

Flat faces are held implicitly in the form

$$Ax + By + Cz + D = 0$$

or in homogeneous co-ordinates it is

$$[x \quad y \quad z \quad 1] \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0$$

Hence flat faces are described by their equations which are held in columns of 4 numbers A, B, C, D.

Under a transformation process T, the point $[x \quad y \quad z \quad 1]$ is transformed into $[x' \quad y' \quad z' \quad 1]$ where $[x' \; y' \; z' \; 1] = [x \; y \; z \; 1] \; T$

So that the transformed flat face equation is

$$\begin{bmatrix} A' \\ B' \\ C' \\ D' \end{bmatrix} = T^{-1} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

T is a 4 x 4 transformation matrix.

2. Curved Faces

Curved faces are held in a parametric form in terms of 2 parameters s and t. There are two types of curved faces in this work.

(a) Cylindrical Faces

The basic cylindrical face is the one shown in Fig. D.1. The form of the parametric equation is

$$[x\ y\ z\ 1] = \left[\frac{t^2}{\Delta(t)} \quad \frac{t}{\Delta(t)} \quad \frac{1}{\Delta t} \quad S\ 1\right] \quad A$$

where

$$\Delta(t) = (2 - \sqrt{2})t^2 + (\sqrt{2} - 2)t + 1$$

and

$$A = \begin{bmatrix} (1 - \sqrt{2}) & (1 - \sqrt{2}) & 0 & 0 \\ (\sqrt{2} - 2) & \sqrt{2} & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

s and t are parameters, both varying from 0 to 1.

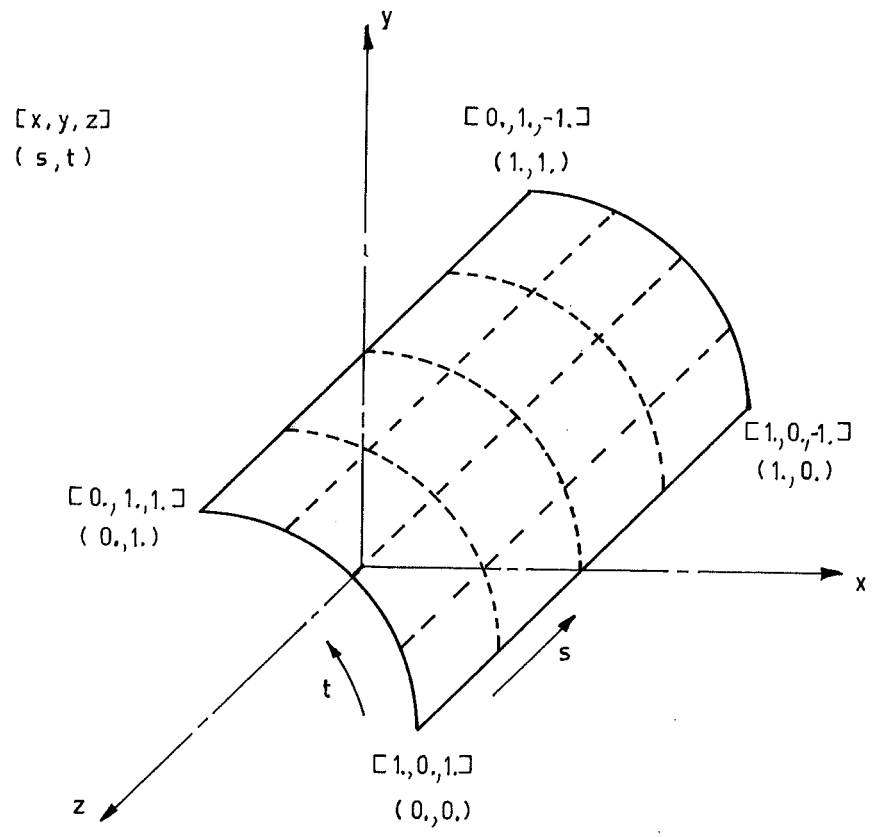


Fig. D.1 Basic Cylindrical Face.

Since the last column of A is always [0 0 0 0 1] to yield the scale factor 1 in the homogeneous co-ordinates, matrix A is stored as a 5 x 3 matrix. Equations of cylindrical faces in the other three quadrants are derived from this basic face by applying a suitable rotation process.

Under any transformation T, it is obvious that the new parametric equation A' is given by

$$A' = AT$$

(b) Conical Faces

This is stored in a similar manner to the case of cylindrical faces. The basic quarter cone is given by:

$$[x \quad y \quad z \quad 1] = [t^2 f \quad tf \quad f \quad s \quad 1] \quad A$$

where

$$f = \frac{(1 - S)}{(2 - \sqrt{2})t^2 + (\sqrt{2} - 2)t + 1}$$

and A is the same as in cylindrical quarter face.

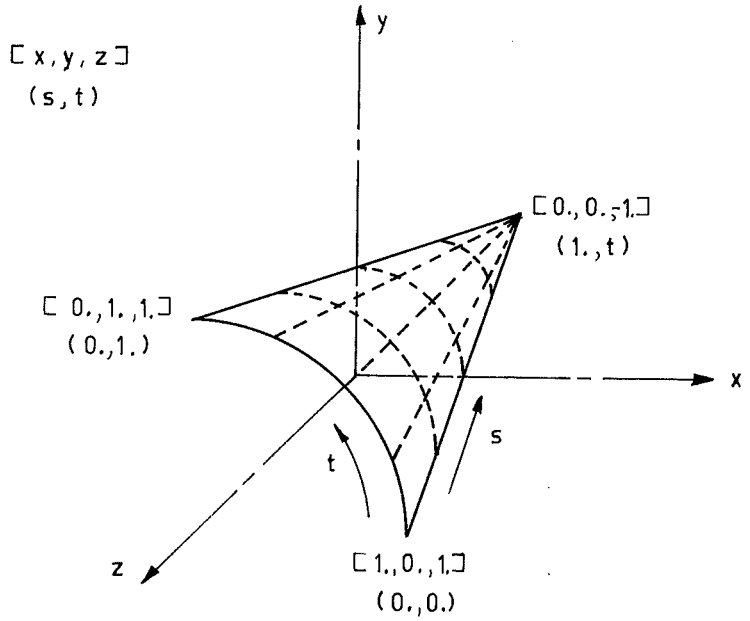


Fig. D.2 The basic quarter cone.

Under any transformation T, the new parametrix matrix A' will be given by

$$A' = AT$$

3. Straight Edges

Straight edges are stored as vertex pairs.

The equation, when required, of a straight edge which has (x_1, y_1, z_1) as its starting point and (x_2, y_2, z_2) as its end point is:

$$[x\ y\ z\ 1] = [x_1\ y_1\ z_1\ 1] + t\,[(x_2 - x_1)\ (y_2 - y_1)\ (z_2 - z_1)\ 0]$$

where $0 \leq t \leq 1$

4. Curved edges lying on cylindrical faces

A curved edge lying on a cylindrical face (either circular or elliptical) is the intersection of a cylindrical face and a flat face not parallel to the cylindrical axis.

Let the flat face be:

$$[x\ \ y\ \ z\ \ 1] \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0$$

and the cylindrical face be:

$$[x\ y\ z\ 1] = \left[\frac{t^2}{\Delta(t)}\ \frac{t}{\Delta(t)}\ \frac{1}{\Delta(t)}\ s\ 1\right] \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix}$$

Solving the 2 equations, we have:

$$s = \left[\frac{t^2}{\Delta(t)}\ \frac{t}{\Delta(t)}\ \frac{1}{\Delta(t)}\ \ 1\right] \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{bmatrix}$$

where

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{bmatrix} = \frac{1}{G} \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

where $G = -(k_{10}A + k_{11}B + k_{12}C)$

Therefore for each edge, the following must be stored:

- the parametric equation of the face on which the edge lies
- the limits of the parameter t at the end points of the curve
- the quadruple (L_1, L_2, L_3, L_4)

5. Curved edges lying on conical faces

Curved edges lying on a conical face are intersections between the conical face and flat faces not passing through the conical axis.

Let the conical face be:

$$[x \quad y \quad z \quad 1] = [t^2 f \quad tf \quad f \quad s \quad 1] \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix}$$

On intersecting with a flat face,

$$s = \frac{L_1 t^2 + L_2 t + L_3}{L_4 t^2 + L_5 t + L_6}$$

where:
$$L_1 = K_1 + (2 - \sqrt{2})\, K_5$$
$$L_2 = K_2 + (\sqrt{2} - 2)\, K_5$$
$$L_3 = K_3 + K_5$$
$$L_4 = K_1 - (2 - \sqrt{2})\, K_4$$
$$L_5 = K_2 - (\sqrt{2} - 2)\, K_4$$
$$L_6 = K_3 - K_4$$

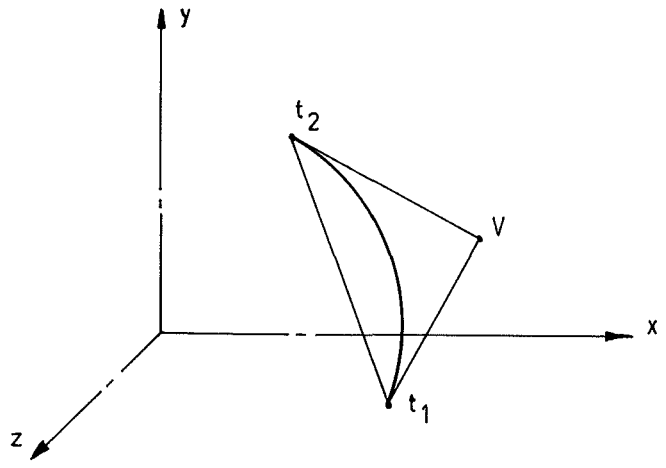
$$\begin{bmatrix} K_1 \\ K_2 \\ K_3 \\ K_4 \\ K_5 \end{bmatrix} = \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

Now each edge is stored in exactly the same manner as the previous cylindrical face case, except that the L numbers now have 6 values (L_1, L_2, L_3, L_4, L_5, L_6) instead of 4.

Some Useful Formulae

1. Tangent Crossing Point:

Intersection detection between a curved edge and another edge (straight or curved) is best performed by enclosing the curved edge with a triangle whose vertices are two end points of the edge and the intersection point of the two tangents at end points; and test to see if the 2nd edge lies within this triangle. If it does not, then the two edges do not intersect. The Point V is called the tangent crossing point.



The tangent crossing point

If the two end points correspond to values $t = t_1$ and $t = t_2$ then the tangent crossing point is found by replacing t^2 by $t_1 t_2$ and t by $(t_1 + t_2)/2$ in the corresponding face equation.

For a cylindrical face, V is given by:

$$[x \quad y \quad z \quad 1]_V = \begin{bmatrix} \frac{t_1 t_2}{\Delta^1(t)} & \frac{t_1+t_2}{2\Delta^1(t)} & \frac{1}{\Delta^1(t)} & S' \quad 1 \end{bmatrix} A$$

where

$$S' = \begin{bmatrix} \frac{t_1 t_2}{\Delta^1(t)} & \frac{t_1 + t_2}{2\Delta^1(t)} & \frac{1}{\Delta^1(t)} & 1 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{bmatrix}$$

and

$$\Delta^1(t) = (2 - \sqrt{2}) t_1 t_2 + (\sqrt{2} - 2) \frac{(t_1+t_2)}{2} + 1$$

and A is the cylindrical face equation matrix.

Similarly for a conical face:

$$[x \; y \; z \; 1]_V = \begin{bmatrix} t_1 t_2 f^1(t) & \frac{(t_1 + t_2) f^1(t)}{2} & f^1(t) \; S' \; 1 \end{bmatrix} A$$

where:

$$S' = \frac{L_1(t_1 t_2) + L_2(t_1+t_2)/2 + L_3}{L_4(t_1 t_2) + L_5 (t_1 + t_2)/2 + L_6}$$

$$f^1(t) = \frac{1 - S'}{(2 - \sqrt{2}) t_1 t_2 + (\sqrt{2} - 2) (t_1 + t_2)/2 + 1}$$

and A is the conical equation matrix.

Comments

To determine whether a straight edge intersects a triangle, it is inadequate to test the positions of just the end points of the edge relative to the triangle, using INSPEC. For example, in Fig. D.3, INSPEC would show that both A and B are outside the triangle, but the edge AB still cuts the triangle.

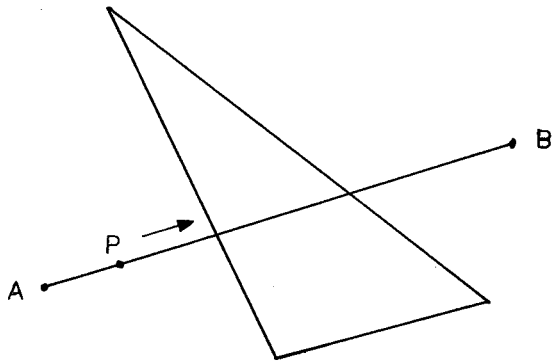


Fig. D.3. Testing an edge and a triangle.

The procedure would be to generate a point P on the edge starting at one end, vertex A, and move it at small equal steps to vertex B. At each position, P is INSPECed and if at any stage P is found to be inside the triangle, the test is stopped and the edge is marked as intersecting. A similar procedure is used when testing a curved edge and a triangle.

2. Flat Face - Flat Face Intersection:

The intersecting line of two non-parallel flat faces

$$[x \quad y \quad z \quad 1]\begin{bmatrix} A_1 \\ B_1 \\ C_1 \\ D_1 \end{bmatrix} = 0 \qquad \text{and} \quad [x \quad y \quad z \quad 1]\begin{bmatrix} A_2 \\ B_2 \\ C_2 \\ D_2 \end{bmatrix} = 0$$

has the direction of the vector $(\underset{\sim}{n}_1 \times \underset{\sim}{n}_2)$ where $\left\{\begin{matrix} \underset{\sim}{n}_1 = A_1\underset{\sim}{i} + B_1\underset{\sim}{j} + C_1\underset{\sim}{k} \\ \underset{\sim}{n}_2 = A_2\underset{\sim}{i} + B_2\underset{\sim}{j} + C_2\underset{\sim}{k} \end{matrix}\right\}$

are the two normal vectors and the point on the line nearest to the origin is:

$$[x_o \; y_o \; z_o] = [-D_1 \quad -D_2 \quad 0] \begin{bmatrix} A_1 & A_2 & B_1C_2 - B_2C_1 \\ B_1 & B_2 & A_2C_1 - A_1C_2 \\ C_1 & C_2 & A_1B_2 - A_2B_1 \end{bmatrix}^{-1}$$

3. Flat Face - Straight Edge intersection

The intersection point of a flat face

$$[A \quad B \quad C \quad D]\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \qquad (1)$$

and a straight finite edge

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} + t \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 0 \end{bmatrix} \quad \text{where } t\varepsilon(0,1) \qquad (2)$$

is given by $t = \dfrac{-(Ax_1 + By_1 + Cz_1 + D)}{(x_2 - x_1)A + (y_2 - y_1)B + (z_2 - z_1)C} = \dfrac{m_1}{m_2}$

If $m_2 = 0$ and $m_1 \neq 0$: the edge is parallel with the face.

$m_2 = 0$ and $m_1 = 0$: the edge lies on the face.

4. Flat Face - Curved Edge Intersection

(a) Curved edge on a Cylindrical Face

Consider a flat face as given in (1) and a cylindrical edge:

$$[x \quad y \quad z \quad 1] = \left[\frac{t^2}{\Delta(t)} \quad \frac{t}{\Delta(t)} \quad \frac{1}{\Delta(t)} \quad S \quad 1\right] \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix}$$

$$\text{where } S = \left[\frac{t^2}{\Delta(t)} \quad \frac{t}{\Delta(t)} \quad \frac{1}{\Delta(t)} \quad 1\right]\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{bmatrix} \qquad (3)$$

where $t\varepsilon(t_1, t_2)$ and $0 \leq t_1 \leq t_2 \leq 1$

$\Delta(t) = Pt^2 + Qt + R$ as before

Substitute (3) into (1) and multiply by $\Delta(t)$:

$$\begin{bmatrix} t^2 & t & 1 & S\Delta(t) & \Delta(t) \end{bmatrix} \begin{bmatrix} K_1 \\ K_2 \\ K_3 \\ K_4 \\ K_5 \end{bmatrix} = 0$$

where $$\begin{bmatrix} K_1 \\ K_2 \\ K_3 \\ K_4 \\ K_5 \end{bmatrix} = \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

Substitute for S and $\Delta(t)$, we have:

$$at^2 + bt + c = 0$$

where: $a = K_1 + K_4 (L_1 + L_4P) + PK_5$

$b = K_2 + K_4 (L_2 + L_4Q) + QK_5$

$c = K_3 + K_4 (L_3 + L_4R) + RK_5$

Therefore, the suitable root of this quadratic gives the intersection point.

(b) Curved Edge on a Conical Face

A similar approach is used and real values of t found in the range $t_1 \leq t \leq t_2$ from the following polynomial correspond to the intersection point:

$$at^4 + bt^3 + ct^2 + dt + e = 0$$

where:

$a = [K_1(L_4 - L_1) + P(K_4L_1 + K_5L_4)]$

$b = [K_1(L_5 - L_2) + K_2(L_4 - L_1) + P(K_4L_2 + K_5L_5) + Q(K_4L_1 + K_5L_4)]$

$c = [K_1(L_6 - L_3) + K_2(L_5 - L_2) + K_3(L_4 - L_1) + P(K_4L_3 + K_5L_6)$

$+ Q(K_4L_2 + K_5L_5) + R(K_4L_1 + K_5L_4)]$

$$d = [K_2(L_6 - L_3) + K_3(L_5 - L_2) + Q(K_4L_3 + K_5L_6) + R(K_4L_2 + K_5L_5)]$$

$$e = [K_3(L_6 - L_3) + R(K_4L_3 + K_5L_6)]$$

t_1, t_2 are values of t which correspond to the end points of the finite curved edge.

The real roots of this polynomial are obtained numerically by the Newton - Raphson iterative method.

5. Curved Face - Straight Edge Intersection

Intersections for a curved face and a straight edge are found from a set of three simultaneous equations in three unknowns s, t, λ. λ is the parameter for the straight edge equation. This set of equations is obtained by equating the expressions for the x, y and z co-ordinates of the curved face and the straight edge. The elimination method could be used to reduce this set to a polynomial in one parameter, say t. The real roots of this polynomial could be solved numerically by the Newton - Raphson method.

Another method was used here which employed a subroutine, called NLNEQN, from the Burroughs subroutine library [4]. Subroutine NLNEQN, developed by Brown, solves numerically a set of non-linear, simultaneous equations. It was used in this application because the elimination method shows to be lengthy and awkward.

Values of s, t and λ are then checked to see if each lies within its allowable limits.

6. Curved Edge - Straight Edge Intersection

Curved edge - straight edge intersections are found by equating the expressions for the x and y co-ordinates, say, of the two edges. This results in a set of two simultaneous equations in two parameters t and λ. λ is further eliminated and a polynomial in t is obtained.

If the curved edge is cylindrical, the polynomial is quadratic and is simple to solve.

If the curved edge is conical, the polynomial is of fourth degree, which can be solved numerically by the Newton-Raphson method.

Values of t and λ are again checked against their allowable limits.

7. Intersection of two coplanar curved edges

(1) Curved edge - Curved edge on Cylindrical faces

Let the two curves be:

$$s = \frac{\ell_1 t^2 + \ell_2 t + \ell_3}{\Delta(t)} + \ell_4$$

$$[x \quad y \quad z \quad 1] = \left[\frac{t^2}{\Delta(t)} \quad \frac{t}{\Delta(t)} \quad \frac{1}{\Delta(t)} \quad s \quad 1\right] \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{15} & 1 \end{bmatrix}$$

where $\Delta(t) = Pt^2 + Qt + 1$

and

$$S = \frac{L_1 T^2 + L_2 T + L_3}{\Delta(2)} + L_4$$

$$[x \quad y \quad z \quad 1] = \left[\frac{T^2}{\Delta(T)} \frac{T}{\Delta(T)} \frac{1}{\Delta(T)} \quad S \quad 1\right] \begin{bmatrix} K_1 & K_2 & K_3 & 0 \\ K_4 & K_5 & K_6 & 0 \\ K_7 & K_8 & K_9 & 0 \\ K_{10} & K_{11} & K_{12} & 0 \\ K_{13} & K_{14} & K_{15} & 1 \end{bmatrix}$$

where $\Delta(T) = PT^2 + QT + 1$

The method involves projecting onto one of the x-y, y-z, and z-x planes, which is not perpendicular to the plane containing the two curved edges, solving for t and T and checking to see if they are in their permissible ranges. This method gives the following results which were quoted incorrectly by Braid [1]:

Solve for t in the quartic equation:

$$u_1 t^4 + u_2 t^3 + u_3 t^2 + u_4 t + u_5 = 0$$

where

$$u_1 = g_1f_1 - [G_1e_1^2 + G_2e_1f_1]$$

$$u_2 = g_1f_2 + g_2f_1 - [2G_1e_1e_2 + G_2(e_1f_2 + e_2f_1)]$$

$$u_3 = g_1f_3 + g_2f_2 + g_3f_1 - [G_1(e_2^2 + 2e_1e_3) + G_2(e_1f_3 + e_2f_2 + e_3f_1) + G_3f_1]$$

$$u_4 = g_2f_3 + g_3f_2 - [2G_1e_2e_3 + G_2(e_2f_3 + e_3f_2) + G_3f_2]$$

$$u_5 = g_3f_3 - [G_1e_3^2 + G_2e_3f_3 + G_3f_3]$$

$$e_1 = \frac{H_1g_1 - G_1h_1 + [G_1H_3 - G_3H_1]f_1}{G_2H_1 - G_1H_2}$$

$$e_2 = \frac{H_1g_2 - G_1h_2 + [G_1H_3 - G_3H_1]f_2}{G_2H_1 - G_1H_2}$$

$$e_3 = \frac{H_1g_3 - G_1h_3 + [G_1H_3 - G_3H_1]f_3}{G_2H_1 - G_1H_2}$$

$$f_1 = \frac{g_1 + dh_1 - aP}{b} \qquad f_2 = \frac{g_2 + dh_2 - aQ}{b} \qquad f_3 = \frac{g_3 + dh_3 - a}{b}$$

where $d = \left[\frac{G_1 + G_2}{H_1 + H_2}\right](-1)$, $a = \frac{G_2 + dH_2}{Q}$, $b = G_3 + dH_3 - \frac{R}{Q}(G_2 + dH_2)$

Now g, h, G, H's are parameters depending on the plane onto which the projection is made. If the projection is made onto the x-y plane, then:

$$\left.\begin{aligned} g_1 &= k_{10}\ell_1 + P(\ell_4k_{10} + k_{13}) + k_1 \\ g_2 &= k_4 + k_{10}\ell_2 + Q(\ell_4k_{10} + k_{13}) \\ g_3 &= k_7 + k_{10}\ell_3 + \ell_4k_{10} + k_{13} \end{aligned}\right\} \text{ from expressions for x}$$

$$\left.\begin{aligned} h_1 &= k_2 + k_{11}\ell_1 + P(\ell_4k_{11} + k_{14}) \\ h_2 &= k_5 + k_{11}\ell_2 + Q(\ell_4k_{11} + k_{14}) \\ h_3 &= k_8 + k_{11}\ell_3 + \ell_4k_{11} + k_{14} \end{aligned}\right\} \text{ from expressions for y}$$

and similar expressions for G_1, G_2, G_3 and H_1, H_2, H_3.

Then

$$T = \frac{e_1t^2 + e_2t + e_3}{f_1t^2 + f_2t + f_3}$$

If the projection is made onto the y - z plane, for example, then

g_1, g_2, g_3 are the expressions for the y co-ordinates and h_1, h_2, h_3 are those for the z co-ordinates which are:

$$h_1 = k_3 + k_{12}\ell_1 + P(\ell_4 k_{12} + k_{15})$$

$$h_2 = k_6 + k_{12}\ell_2 + Q(\ell_4 k_{12} + k_{15})$$

$$h_3 = k_9 + k_{12}\ell_3 + \ell_4 k_{12} + k_{15}$$

2. Curved Edge - Curved Edge on Conical Faces

Let the 2 curved edges be:

$$s = \frac{\ell_1 t^2 + \ell_2 t + \ell_3}{\ell_4 t^2 + \ell_5 t + \ell_6}$$

$$[x \quad y \quad z \quad 1] = [t^2 f \quad tf \quad f \quad s \quad 1] \begin{bmatrix} k_1 & k_2 & k_3 & 0 \\ k_4 & k_5 & k_6 & 0 \\ k_7 & k_8 & k_9 & 0 \\ k_{10} & k_{11} & k_{12} & 0 \\ k_{13} & k_{14} & k_{14} & 1 \end{bmatrix}$$

where $f = \dfrac{1 - s}{Pt^2 + Qt + 1}$

and $S = \dfrac{L_1 T^2 + L_2 T + L_3}{L_4 T^2 + L_5 T + L_6}$

$$[x \quad y \quad z \quad 1] = [T^2 F \quad TF \quad F \quad S \quad 1] \begin{bmatrix} K_1 & K_2 & K_3 & 0 \\ K_4 & K_5 & K_6 & 0 \\ K_7 & K_8 & K_9 & 0 \\ K_{10} & K_{11} & K_{12} & 0 \\ K_{13} & K_{14} & K_{15} & 1 \end{bmatrix}$$

where $F = \dfrac{(1 - S)}{PT^2 + QT + 1}$

t and T are found from the set on non-linear equations:

$$\frac{g_1t^4 + g_2t^3 + g_3t^2 + g_4t + g_5}{r_1t^4 + r_2t^3 + r_3t^2 + r_4t + r_5} = \frac{G_1T^4 + G_2T^3 + G_3T^2 + G_4T + G_5}{R_1T^4 + R_2T^3 + R_3T^2 + R_4T + R}$$

$$\frac{h_1t^4 + h_2t^3 + h_3t^2 + h_4t + h_5}{r_1t^4 + r_2t^3 + r_3t^2 + r_4t + r_5} = \frac{H_1T^4 + H_2T^3 + H_3T^2 + H_4T + H_5}{R_1T^4 + R_2T^3 + R_3T^2 + R_4T + R_5}$$

with the help of the Burroughs library subroutine NLNEQN.

Values found for t, T are then checked to see if they are in the right ranges.

Where:

$$g_1 = (\ell_4 - \ell_1)k_1 + P(k_{10}\ell_1 + k_{13}\ell_4)$$

$$g_2 = (\ell_4 - \ell_1)k_4 + (\ell_5 - \ell_2)k_1 + P(k_{10}\ell_2 + k_{13}\ell_5) + Q(k_{10}\ell_1 + k_{13}\ell_4)$$

$$g_3 = (\ell_4 - \ell_1)k_7 + (\ell_5 - \ell_2)k_4 + (\ell_6 - \ell_3)k_1 + P(k_{10}\ell_3 + k_{13}\ell_6) + Q(k_{10}\ell_2 + k_{13}\ell_5) + k_{10}\ell_1 + k_{13}\ell_4)$$

$$g_4 = (\ell_5 - \ell_2)k_7 + (\ell_6 - \ell_3)k_4 + Q(k_{10}\ell_3 + k_{13}\ell_6) + (k_{10}\ell_2 + k_{13}\ell_5)$$

$$g_5 = (\ell_6 - \ell_3)k_7 + (k_{10}\ell_3 + k_{13}\ell_6)$$

Expressions for h_1, h_2, h_3, h_4, h_5 are exactly the same as above when column $[k_1, k_4, k_7, k_{10}, k_{13}]$ is replaced by column $[k_2, k_5, k_8, k_{11}, k_{14}]$.

$$r_1 = P\ell_4$$

$$r_2 = Q\ell_4 + P\ell_5$$

$$r_3 = \ell_4 + Q\ell_5 + P\ell_6$$

$$r_4 = \ell_5 + Q\ell_6$$

$$r_5 = \ell_6$$

Expressions for G_1, G_2, G_3, G_4, G_5, H_1, H_2, H_3, H_4, H_5, R_1, R_2, R_3, R_4, R_5 are similar to the above.

3. Curved Edge - Curved edge: one on Cylindrical face, one on Conical Face

Let the "cylindrical edge" be defined in lower case letters and the "conical edge" be defined in upper case letters as above. Also a similar method is employed here. t and T are found by solving the 2 non-linear equations:

$$(x)_{cyl.} = (x)_{con.}$$

$$(y)_{cyl.} = (y)_{con.}$$

i.e.

$$\frac{g_1t^2 + g_2t + g_3}{Pt^2 + Qt + 1} = \frac{G_1T^4 + G_2T^3 + G_3T^2 + G_4T + G_5}{R_1T^4 + R_2T^3 + R_3T^2 + R_4T + R_6}$$

$$\frac{h_1t^2 + h_2t + h_3}{Pt^2 + Qt + 1} = \frac{H_1T^4 + H_2T^3 + H_3T^2 + H_4T + H_5}{R_1T^4 + R_2T^3 + R_3T^2 + R_4T + R_5}$$

where the coefficients are as previously defined.

8. Intersection of 2 straight edges

This is fully described in Appendix E which also shows how the relative position of the intersection point with respect to the two edges is described. The description of the position of the intersection point is exactly the same when one or both edges are curved.

9. The Normal Vector of a Curved Face

For a curved face, the expressions (in terms of the parameters s and t) for the co-ordinates of a point lying on the face are:

$$x = x(s, t)$$

$$y = y(s, t)$$

$$z = z(s, t)$$

The normal vector $\underset{\sim}{n}$ of the face at a point where $s = s_o$ and $t = t_o$ is:

$$\underset{\sim}{n} = \underset{\sim}{n}_s \times \underset{\sim}{n}_t$$

where $\underset{\sim}{n}_s = \left(\frac{\delta x(s,\, t)}{\delta s}, \frac{\delta y(s,\, t)}{\delta s}, \frac{\delta z(s,\, t)}{\delta s}\right)_{(s_o,\, t_o)}$

$\underset{\sim}{n}_t = \left(\frac{\delta s(s,\, t)}{\delta t}, \frac{\delta y(s,\, t)}{\delta t}, \frac{\delta z(s,\, t)}{\delta t}\right)_{(s_o,\, t_o)}$

APPENDIX E

INTERSECTION OF TWO FINITE STRAIGHT EDGES

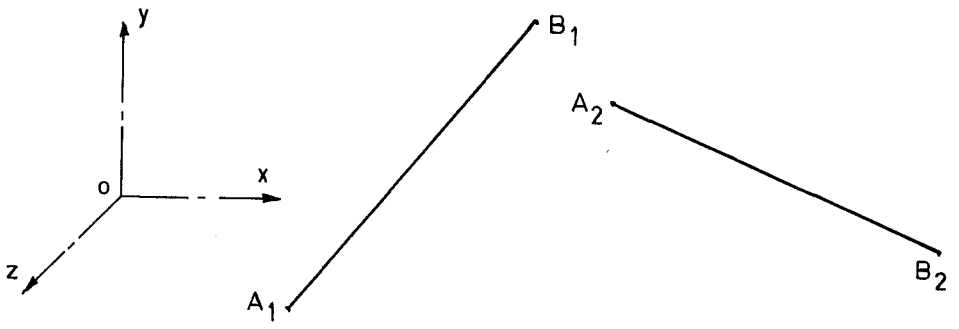
1. INTRODUCTION

The point of intersection of two straight line segments in space is usually obtained by expressing each line in parametric form, P $(x(t_1), y(t_1), z(t_1))$ and P $(x(t_2), y(t_2), z(t_2))$ where t_1 and t_2 are parameters in the range $[0,1]$. If an intersection exists the following three equations with two unknowns t_1 and t_2 are satisfied.

$$x(t_1) = x(t_2)$$
$$y(t_1) = y(t_2)$$
$$z(t_1) = z(t_2)$$

The procedure to find the intersection is to solve two equations for t_1 and t_2, if an intersection exists these values will satisfy the third equation. A more efficient procedure is described here which involves the use of dot and cross products of vectors.

2. THE METHOD OF DESCRIBING THE INTERSECTION OF TWO LINE SEGMENTS



Equation of a point P on the line segment A_1B_1 is

$$\overline{OP} = \overline{OA_1} + t_1\overline{A_1B_1} \quad \text{where } t_1 \varepsilon [0,1] \text{ - (1)}$$

Similarly for a point P on line segment A_2B_2

$$\overline{OP} = \overline{OA_2} + t_2\overline{A_2B_2} \qquad - (2)$$

where $t_2 \varepsilon [0,1]$

Thus, if an intersection exists,

$$\overline{OA_1} + t_1\overline{A_1B_1} = \overline{OA_2} + t_2\overline{A_2B_2} \qquad - (3)$$

which in terms of the x, y, z components is

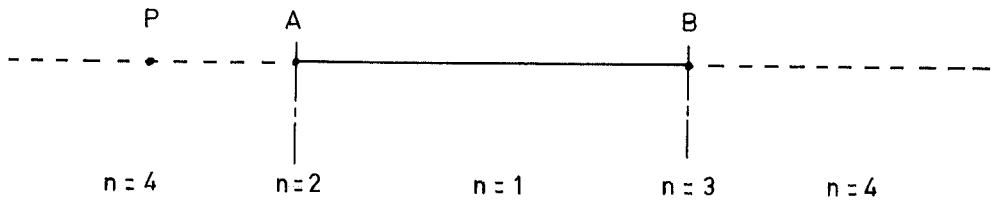
$$x_1(t_1) = x_2(t_2) \qquad - (4)$$

$$y_1(t_1) = y_2(t_2) \qquad - (5)$$

$$z_1(t_1(= z_2(t_2) \qquad - (6)$$

The usual procedure is to project the 2 lines on to a plane which is not perpendicular to the plane containing the two lines, for example, the x-y plane. Then equations (4) and (5) are used to solve for t_1 and t_2. The obtained values of t_1 and t_2 are then substituted into (6) to see if they satisfy (6). If (6) is not satisfied the intersection does not exist. If (6) is satisfied then the values of t_1 and t_2 are checked to see if they are outside the range $[0,1]$. If they are, the two line segments do not intersect.

Following the work of Braid , the position of a point P on the line AB relative to the segment AB is described in terms of a function n (t) defined as follows:



n(t) = 1 if t is such that +d <t <1 - d where t is again given by

$$\overline{OP} = \overline{OA} + t\overline{AB}$$

i.e., P is inside segment AB

d is a small number, practically d = 0.00001

$n(t) = 2$ if $-d < t < d$

i.e., P coincides with A

$n(t) = 3$ if $1 - d < t < 1 + d$

i.e., P coincides with B

$n(t) = 4$ if $1 + d < t$

or $t < -d$

i.e., P lies outside segment AB

The intersection of 2 non-colinear segments A_1B_1 and A_2B_2 is thus described by a pair (n_1, n_2) where $n_1 = n(t_1)$ and $n_2 = n(t_2)$.

When two segments A_1B_1 and A_2B_2 are colinear, there is of course, no unique intersection and the position of the two lines relative to each other is determined by establishing the position of each of the two end points of one line segment relative to the other line segment. Thus, a quadruple (n_1, n_2, n_3, n_4) is formed where the values of n_1 and n_3 give the positions of points A_2 and B_2 respectively on the line A_1B_1. Similarly, the values of n_2 and n_4 give the positions of A_1 and B_1 on line A_2B_2 respectively. In mathematical terms, they are defined as:

$n_1 = n(t_1)$ where t_1 is obtained by evaluating

$$\overline{OA_2} = \overline{OA_1} + t_1\overline{A_1B_1}$$

$n_2 = n(t_2)$ where t_2 is obtained by evaluating

$$\overline{OA_1} = \overline{OA_2} + t_2\overline{A_2B_2}$$

$n_3 = n(t_3)$ where t_3 is obtained by evaluating

$$\overline{OB_2} = \overline{OA_1} + t_3\overline{A_1B_1}$$

$n_4 = n(t_4)$ where t_4 is obtained by evaluating

$$\overline{OB_1} = \overline{OA_2} + t_4\overline{A_2B_2}$$

Finally, a number n_5 is included to give the number of intersections of the two lines. It can be 0, 1 or 2 for lines having no intersection, one intersection or which are colinear, respectively. Therefore, for any two general lines, a quintuple $(n_1, n_2, n_3, n_4, n_5)$ is stored and it is accessed by the computer in the following manner:

1. If $n_5 = 0$: the computer ignores n_1, n_2, n_3, n_4.
2. If $n_5 = 1$: the computer accesses only the values of n_1 and n_2.
3. If $n_5 = 2$: the computer accesses all values of n_1, n_2, n_3 and n_4.

The condition for an intersection to occur is that all 4 points A_1, B_1, A_2, B_2 are coplanar, i.e., $(\overline{A_1B_1} \times \overline{A_2B_2}) . \overline{A_1A_2} = 0$

If this is satisfied then cross product equation (3) with $\overline{A_2B_2}$ to give

$$(\overline{OA_1} \times \overline{A_2B_2}) + t_1(\overline{A_1B_1} \times \overline{A_2B_2}) = (\overline{OA_2} \times \overline{A_2B_2}) \quad - \quad (7)$$

Then dot product with $\overline{OA_2}$ to give:

$$(\overline{OA_1} \times \overline{A_2B_2}) . \overline{OA_2} + t_1(\overline{A_1B_1} \times \overline{A_2B_2}) . OA_2 = 0$$

therefore:

$$t_1 = - \frac{(\overline{OA_1} \times \overline{A_2B_2}) . \overline{OA_2}}{(\overline{A_1B_1} \times \overline{A_2B_2}) . \overline{OA_2}}$$

Then t_1 is put into (1) to find $\overline{OP}$.

Now the position of P relative to each segment is determined by a call to a subroutine TESTPL, which works as follows:

Form 2 vectors

$\overline{PA} = \overline{OA} - \overline{OP}$

$\overline{PB} = \overline{OB} - \overline{OP}$

If $\overline{PA} = \overline{0}$ (shown by $\overline{PA} \cdot \overline{PA} < d$) then $n = 2$

If $\overline{PB} = \overline{0}$ then $n = 3$

If $(\overline{PA} \cdot \overline{PB}) < 0$: $n = 1$

If $(\overline{PA}\cdot\overline{PB}) > 0$: n = 4

And because subroutines to obtain cross product and dot product of two vectors are easily written, this method proves itself to be very neat and efficient.

3. THE PROCEDURE FOR DETERMINING THE INTERSECTION OF TWO LINE SEGMENTS

1. Form three vectors:

$$\overline{A_1B_1} = \overline{OB_1} - \overline{OA_1}$$

$$\overline{A_2B_2} = \overline{OB_2} - \overline{OA_2}$$

$$\overline{A_1A_2} = \overline{OA_2} - \overline{OA_1}$$

2. Form vector $\overline{U} = \overline{A_1B_1} \times \overline{A_2B_2}$

If $\overline{U}\cdot\overline{U} < d$, then go to step 10.

3. Form vector $\overline{V} = \overline{U}\cdot\overline{A_1A_2}$

If $\overline{V}\cdot\overline{V} < d$, then go to step 13.

4. Obtain number p_2 where

$P_2 = \overline{U}\cdot\overline{OA_2}$ and set $n_5 = 1$

If absolute value of p_2, $|p_2| < d$, go to step 9.

5. Obtain number p_1 where

$$p_1 = (\overline{OA_1} \times \overline{A_2B_2}) \cdot \overline{OA_2}$$

6. Get : $t_1 = -p_1/p_2$

7. Obtain vector $\overline{OP}$

$$\overline{OP} = \overline{OA_1} + t_1\overline{A_1B_1}$$

8. Call TESTPL to obtain n_1 and n_2.

The procedure is completed.

9. Recall that at this stage we have equation (7) which is

$$(\overline{OA_1} \times \overline{A_2B_2}) + t_1(\overline{A_1B_1} \times \overline{A_2B_2}) = (\overline{OA_2} \times \overline{A_2B_2})$$

and we cannot dot product it with $\overline{OA_2}$ because this will give a zero coefficient for t_1.

Form 3 numbers:

$$p_1 = (\overline{OA_1} \times \overline{A_2B_2}) \cdot \overline{U}$$

$$P_2 = \overline{U} \cdot \overline{U}$$

$$p_3 = (\overline{OA_2} \times \overline{A_2B_2}) \cdot \overline{U}$$

Since p_2 cannot be zero, get

$$t_1 = \frac{p_3 - p_1}{p_2}$$

Go to step 7.

10. Now A_1B_1 and A_2B_2 are either parallel or colinear.

If $\overline{A_1A_2} \cdot \overline{A_1A_2} < d$, go to step 12.

Obtain vector $\overline{W}$ where

$$\overline{W} = \overline{A_1A_2} \times \overline{A_1B_1}$$

If $\overline{W} \cdot \overline{W} < d$, go to step 12.

11. Two lines $\overline{A_1B_1}$ and $\overline{A_2B_2}$ are now parallel.

Go to step 13.

12. $\overline{A_1B_1}$ and $\overline{A_2B_2}$ are now colinear.

Set $n_5 = 2$.

Call TESTPL to obtain n_1, n_2, n_3, n_4.

The procedure is completed.

13. There cannot be any intersection because the two lines A_1B_1 and A_2B_2 are either parallel or non-planar.

Set $n_5 = 0$

Procedure is completed.

4. CONCLUSION

The procedure described above provides an efficient method for determining the intersection of two straight line segments. It overcomes all of the difficulties that may arise in solving 3 equations for 2 unknowns.

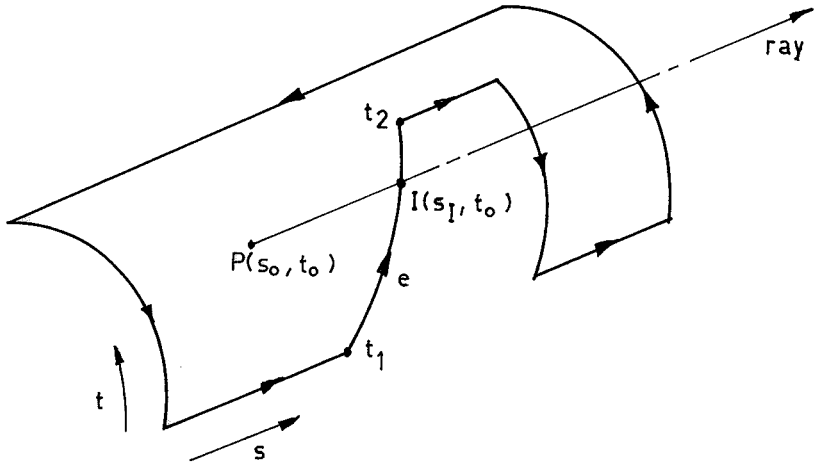
APPENDIX F

SUBROUTINE CINSPE

Subroutine CINSPE, like subroutine INSPEC, determines whether a point P lying on a face is inside or outside the face bounded by its edges. INSPEC applies to a flat face and CINSPE applies to a curved face.

Both routines were fully described in Braid's work. However, in routine CINSPE, there were details which showed themselves inefficient and subsequent changes were made and described below.

Consider a point $P(s_o, t_o)$ lying on a curved face as shown in the figure below:



The method involves passing a ray $t = t_o$, $s \geq s_o$ from the point P and all intersections between this ray and the edges of the curved face are recorded. The final record (of intersections) determines whether P is inside or outside the face.

The procedure is:

1. Test to see if either s_o or t_o lies outside the range (0, 1). If so, P lies outside the face. Otherwise, set IC, the intersection counter equal zero.

2. Get the next edge e belonging to the face. If there are no more edges left, go to step 7.

3. If e is a straight edge, e cannot cross the ray since a straight edge must always be either parallel to or colinear with the ray. Go to step 2 without altering IC.

4. The edge e is curved. Let t_1 and t_2 be the parameter limits of the edge e. If t_o lies outside the range (t_1, t_2), go to step 2. If $t_o = t_1$ or $t_o = t_2$ go to step 5. Otherwise the edge may cross the ray.

   At this point, Braid suggested that the values s_1 and s_2 at the end points of edge e, which corresponded to t_1 and t_2 be calculated and compared with s_o. If both $s_1 \gtrless s_o$ and $s_2 \gtrless s_o$ an intersection has occurred. Otherwise, tests had to be made at the end points and many different cases had to be distinguished. There was also a possibility that one had to solve a quadratic equation to find exactly where the intersection was. It was at this point that changes were made as follows:

   Let I be the intersection point of the line (not the ray) $t = t_o$ and the edge e. At I, $t = t_o$ and $s = s_I$ which can be easily calculated from t_o. If $s_I < s_o$, I is not on the ray and go to step 2. If $s_I = s_o$, P lies on the edge e, exit from the routine. Otherwise $s_I > s_o$, I is on the ray, go to step 6.

5. P must lie on the line containing the straight edge that either precedes or follows the current curved edge e. Get this straight edge. If P lies on this straight edge, P is on the face and exit from the routine. Otherwise, go to step 2.
   If there is not a straight edge preceding or following the current curved edge e, test if P is one of the end vertices of e, i.e. $s_I = s_1$ (if $t = t_1$) or $s_I = s_2$ (if $t = t_2$). If this is so, exit from the routine. Otherwise go to step 2.

6. IC is increased by +2 or -2 depending on the direction in which the ray crosses the edge. Go to step 2.

7. The absolute value of IC is divided by 4 and the remainder is kept. If the remainder is 0, P lies outside the face. If it is 2, P is inside.

   The routine is complete.

REFERENCES

1. BRAID, I.C., "Designing with volumes", Ph.D. Thesis 8404, Cambridge University, 1974.

2. COONS, S.A., "Surfaces for computer-aided design of space forms", U.S. Government Report No. AD 663504, June 1967.

3. WELBOURN, D.B., "Computer graphics in design and manufacture", pp 67-71, C.M.E. Jnl, 1975.

4. B6700NSL - Numerals Numerical Analysis Program Library, Division of Boroughs Corporation, Detroit, Michigan, 48232.

5. IBM SYSTEM/360 - Scientific Subroutine Package, 360A-CM-03X, VERSION III Programmer's Manual.

6. RUBIN, IRVIN I., "Injection moulding theory and practice", Wiley, New York, 1973.

7. FORREST, A.R., "Curves and surfaces for computer-aided design", Ph.D. Thesis, Univ. of Cambridge, England, 1968.

8. GOSSLING, T.H., "Computer-aided redesign of a cast component", C.A.D. Journal, Vol.7, No.3, July 1975, pp 198-199.

9. CLARK, JAMES H., "Designing surfaces in 3-dimensions", Communications of the A.C.M. Vol.19, No.8, August 1976, pp 454-459.

10. SADEGHI, M.M. and GOULD, S.S., "A comparison of 2 parametric surface patch methods", C.A.D. Journal, Vol.6, No.4, October 1974, pp 217-220.

11. GLANVILL, ALAN BIRKETT and DENTON, E.N., "Injection - Mould Design Fundamentals", New York Industrial Press, 1965.

12. GLANVILL, A.B., "Plastics Engineers' data book", The Machinery Publishing Co. Ltd, 1971.

13. BUTLER, J., "Compression and transfer moulding of plastics", Interscience Publisher Inc., 1959.

14. MILBY, R.V., "Plastics technology", McGraw-Hill Book Co., 1973.

15. BECK, R.D., "Plastic product design", Van Nostrand Reinhold Co., 1970.

16. SORS LÁSZ LO, "Plastic mould engineering", The English Edition, 1967, published by Akadémiai Kiadó, Budapest.

17. BUTLER, J., "Compression and transfer moulding of plastics", Iliffe & Sons Ltd, 1959.

18. DUBOIS, J.H., and PRIBBLE, W.I., "Plastics mould engineering", S.P.E. Polymer Technology Series, Van Nostrand Reinhold Co., 1965.

19. RT-11 System Reference Manual, DEC-11-ORUGA-C-DN1, 1975.

20. RT-11 Fortran Compiler and Object Time System, User's Manual. DEC-11-LRFPA-A-D, 1974.

21. PDP-11 Fortran Language Reference Manual, DEC-11-LFLRA-B-D, 1974.

22. Fortran RT-11 Extensions Manual, DEC-11-LRTEA-B-D, 1975.

23. BOLTON, K.M., "Biarc Curves", Computer-aided design, Vol.7, No.2, April 1975, pp 89-92.

24. SPACEK, THOMAS R., "A proposal to establish a pseudo virtual memory via writable overlays", Communications of the A.C.M., Vol.15, No.6, June 1972, pp 421-426.

25. BAECKER, H.D., "Garbage collection for virtual memory computer systems", Communications of the A.C.M., Vol.15, No.11, Nov.1972, pp 981-986,

26. DENNING, PETER J., "Virtual memory", Computer Surveys, Vol.2, No.3, September 1970, pp 153-189.

27. PANKHURST, R.J., "Program overlay techniques", Communications of the A.C.M., Vol.11, No.2, Feb.1968, pp 119-125.

28. AHUJA, D.V., and COONS, S.A., "Geometry for construction and display", I.B.M. Systems Journal Nos 3 and 4, 1968, pp 188-205.

29. RICCI, A., "A constructive geometry for computer graphics", The Computer Journal, Vol.16, No.2, pp 157-160.

30. COHEN, DAN and LEE, THEODORE M.P., "Fast drawings of curves for computer display", Spring Joint Computer Conference, 1969, pp 297-307.

31. LEE, THEODORE M.P. "A class of surfaces for computer display", Spring

Joint Computer Conference, 1969, pp 309-319.

32. DZUBAK, B.J. and WARBURTON, C.R., "The organisation of structured files", Communications of the A.C.M., Vol.8, No.7, July 1965, pp 446-452.

33. BLAKE, LINDEN F., LAWSON ROSEMARIE E., and YUILLE, I.M., "A ring processing package for use with FORTRAN or a similar high-level language", The Computer Journal, Vol.13, No.1, Feb.1970, pp 40-47.

34. CHRISTOFFERSON, JOHN L., "Mini strings with many uses", DATAMATION, Dec.1973, pp 87-92.

35. BORROW, DANIEL G., and MURPHY, DANIEL L., "Structure of a LISP system using two-level storage", Communications of the A.C.M., Vol.10, No.3, Mar.1967, pp 155-159.

36. CASEY, R.G., "Design of tree structures for efficient querying", Communications of the A.C.M., Vol.16, No.9, Sept.1973, pp 549-556.

37. CLAMPETT, HARRY A. JR., "Randomised binarry searching with tree structures", Communications of the A.C.M., Vo..7, No.3, Mar.1964, pp 163-165.

38. VAN DAM, ANDRIES, and EVANS, DAVID, "A compact data structure for storing, retrieving and manipulating line drawings", Spring Joint Computer Conference, 1967, pp 601-610.

39. SEXTON, J.H., "An introduction to data-structures with some emphasis on graphics", The Computer Bulletin, Sept. 1972, pp 444-447.

40. IRANI, K.B. and JACKSON, J.H., "An approach to the optimum implementation of interactive display data structures", Computer Graphics and Image Processing (1972), 1, pp 221-243.

41. DODD, GEORGE G., "Elements of data management systems", Computing Surveys, Vol.1, No.2, June 1969.

42. WILLIAMS, ROBIN, "A survey of data structures for computer graphics systems", Computing Surveys, Vol.3, No.1, March 1971, pp 1-21.

43. LANG, C.A., and GRAY, J.C., "ASP - A ring implemented associative structure package", Communications of the A.C.M., Vol.11, No.8, Aug.1968, pp 550-555.

44. GRAY, J.C., "Compound data structure for computer aided design; a survey", Proc. A.C.M. 20th Nat.Conf., 1967, pp 355-365.

45. EARLEY, JAY, "Toward an understanding of data structures", Communications of the A.C.M., Vol.14, No.10, Oct.1971, pp 617-627.

46. GUZMAM, A., "Decomposition of a visual scene into 3-dimensional bodies", Automatic Interpretation and Classification of Images - Academic Press, New York, 1969, pp 243-276.

47. GOURAND, HENRI, "Computer display of curved surfaces", June 1971, UTEC-CSc-71-113, Computer Science, Univ. of Utah, Salt Lake City, Utah, 84112.

48. WATKINS, GARY SCOTT, "A real-time visible surface algorithm", June 1970 UTEC-CSc-70-101, Computer Science, Univ. of Utah, Salt Lake City, Utah 84112.

49. BASKIN, H.B., and MORSE, S.P., "A multi-level modelling structure for interactive graphic design", I.B.M. Systems Journal 7, 1968, pp 218-228.

50. MARTIN, WILLIAM A., "Sorting", Computing Surveys Vol.3, No.4, Dec.1971, pp 147-174.

51. COALES, J.F., "Computer-aided design - the common ground", Computer-Aided Design, Vol.5, No.2, 1973, pp 106-108.

52. LEWIS, W.P., SAMUEL, A.E., and FIELD, B.W., "An example of the application of a systematic method to design", Oper. Res. Q., Vol.24, No.2, June 1973, pp 217-233.

53. BAUMGART, B.G., "Geometric modeling for computer vision", N.T.I.S. U.S. Dept. of Commerce, AD-A002-261, Oct. 1974.

54. BAUMGART, B.G., "A polyhedron representation for computer vision:, National Computer Conference, 1975, pp 589-596.

55. GORDON, WILLIAM J., and RIESENFIELD, RICHARD F., "Bernstein-Bézier methods for the computer-aided design of free-form curves and surfaces", Journal of the A.C.M., Vol.21, No.2, Apr.1974, pp 293-310.

56. SPUR, G., and GAUSEMEIER, J., "Processing of workpiece information for producing engineering drawings", Proc. of the 16th International Machine Tool Design and Research Conference, Edited by F. KOENIGSBERGER and S.A. TOBIAS, The MacMillan Press Ltd, 1976, pp 29-33.

57. MORRIS, R.B. and WELBOURN, D.B., "Computer graphics and numerically controlled machine tools for pattern, mould, and die production", Proc. of the 16th International Machine Tool Design and Research Conference. Edited by F. KOENIGSBERGER and S.A. TOBIAS. The MacMillan Press Ltd, 1976, pp 155-159.

58. BRAID, I.C., "The synthesis of solids bounded by many faces", Communications of the A.C.M., Vol.18, No.4, Apr. 1975, pp 209-216.

59. ADAMS, J.A., "Geometric concepts for computer graphics", U.S. Dept. of Commerce, AD-750 743, Sept. 1972.